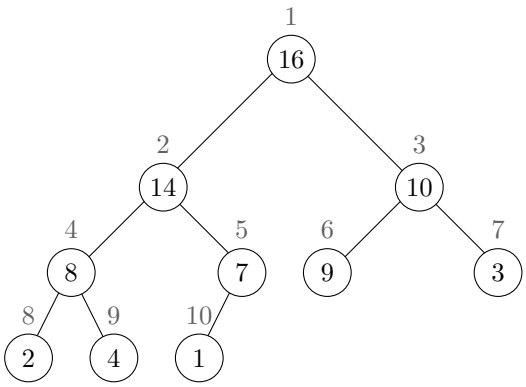
# Algorithmentechnik

Skript zur Vorlesung
von
Prof. Dorothea Wagner,
Karlsruhe, Wintersemester 09/10
Stand: 16. Oktober 2009




Skript erstellt von

Robert Görke und Steffen Mecke
Fakultät für Informatik
Karlsruher Institut für Technologie

Basierend auf

den Vorlesungsnotizen von Dorothea Wagner
und dem Skript der Vorlesung
„Algorithmen und Datenstrukturen“ aus Konstanz

# Inhaltsverzeichnis

**Inhaltsverzeichnis** **1**

**Abbildungsverzeichnis** **5**

**Algorithmenverzeichnis** **7**

**0 Grundlagen** **1**

0.1 Worst-case- und Average-case-Laufzeit . . . 1

0.1.1 Beispiel: Worst-case-Laufzeit von Finde Maximum . . . 1

0.1.2 Beispiel: Average-case-Laufzeit von Finde Maximum . . . 2

0.1.3 Beispiel: Average-case-Laufzeit von Quicksort . . . 3

0.2 Untere Schranken für Laufzeiten . . . 4

0.2.1 Untere Schranke für Sortieren . . . 4

0.2.2 Untere Schranke für Berechnung der konvexen Hülle . . . 4

0.3 Amortisierte Analyse . . . 5

0.3.1 Beispiel von Stackoperationen . . . 6

0.3.2 Die Ganzheitsmethode . . . 7

0.3.3 Die Buchungsmethode . . . 7

0.3.4 Die Potentialmethode . . . 7

0.4 Divide-and-Conquer Verfahren und das Prinzip der Rekursion . . . 9

0.4.1 Ein Beispiel aus der Informatik-Grundvorlesung . . . 9

0.4.2 Rekursionsabschätzungen . . . 10

0.4.3 Die Substitutionsmethode . . . 11

0.4.4 Die Iterationsmethode . . . 12

0.4.5 Der Aufteilungs-Beschleunigungssatz (Meistermethode, Master-Theorem) . . . 12

**1 Grundlegende Datenstrukturen für Operationen auf Mengen** **19**

1.1 Union-Find . . . 19

1.1.1 Drei Ansätze . . . 19

1.1.2 Die Laufzeit von Union-Find . . . 24

1.1.3 Bemerkungen . . . 28

1.2 Anwendungsbeispiele für Union-Find . . . 29

1.2.1 Der Algorithmus von Kruskal für MST . . . 29
1.2.2 Das Offline-Min-Problem . . . 29
1.2.3 Äquivalenz endlicher Automaten . . . 30
1.3 Priority Queues oder Heaps . . . 34
1.3.1 HEAPIFY . . . 35
1.3.2 MAKEHEAP . . . 36
1.3.3 Prozedur DELETE . . . 37
1.3.4 Prozedur SIFT-UP . . . 38
1.3.5 Prozedur INSERT . . . 38
1.3.6 Sortierverfahren HEAPSORT . . . 39
1.3.7 Alternative BOTTOM-UP-HEAPIFY . . . 39

**2 Aufspannende Bäume minimalen Gewichts 41**
2.1 Einführung . . . 41
2.2 Das MST-Problem . . . 41
2.2.1 Motivation . . . 42
2.3 Die Färbungsmethode von Tarjan . . . 42
2.3.1 Grüne Regel . . . 43
2.3.2 Rote Regel. . . . 43
2.3.3 Färbungsinvariante . . . 44
2.4 Der Algorithmus von Kruskal . . . 46
2.5 Der Algorithmus von Prim . . . 47
2.5.1 Implementation des Algorithmus von Prim . . . 47
2.6 Greedy–Verfahren und Matroide . . . 49

**3 Schnitte in Graphen und Zusammenhang 53**
3.1 Schnitte minimalen Gewichts: MINCUT . . . 53
3.2 Der Algorithmus von Stoer & Wagner . . . 55

**4 Flussprobleme und Dualität 63**
4.1 Grundlagen . . . 63
4.1.1 Problemstellung . . . 64
4.2 Bestimmung maximaler Flüsse („Max Flow“) . . . 66
4.2.1 Ford-Fulkerson-Algorithmus (1962) . . . 68
4.2.2 Der Algorithmus von Edmonds und Karp (1972) . . . 68
4.2.3 Der Algorithmus von Goldberg und Tarjan (1988) . . . 72
4.2.4 Anwendungsbeispiel: Mehrmaschinen-Scheduling . . . 81

**5 Kreisbasen minimalen Gewichts 83**
5.1 Kreisbasen . . . 83
5.2 Das Kreismatroid . . . 85

5.3 Der Algorithmus von Horton . . . 85
5.4 Der Algorithmus von de Pina (1995) . . . 87
5.4.1 Beispiel zum Algorithmus von de Pina . . . 88
5.4.2 Korrektheit des Algorithmus von de Pina . . . 89
5.5 Effiziente Berechnung eines Zertifikats für MCB . . . 91

**6 Lineare Programmierung 93**
6.1 Geometrische Repräsentation von LPs . . . 95
6.2 Algorithmen zur Lösung von LPs . . . 97

**7 Approximationsalgorithmen 99**
7.1 Approximationsalgorithmen mit relativer Gütegarantie . . . 100
7.1.1 Das allgemeine Knapsack Problem . . . 100
7.1.2 Bin Packing (Optimierungsversion) . . . 101
7.2 Approximationsschemata . . . 105
7.2.1 Ein PAS für Multiprocessor Scheduling . . . 106
7.3 Asymptotische PAS für Bin Packing . . . 109
7.3.1 Ein APAS für Bin Packing . . . 109
7.3.2 AFPAS für Bin Packing . . . 114

**8 Randomisierte Algorithmen 117**
8.1 Grundlagen der Wahrscheinlichkeitstheorie I . . . 118
8.2 Randomisierte MinCut-Algorithmen . . . 119
8.2.1 Ein einfacher Monte Carlo-Algorithmus für MinCut . . . 119
8.2.2 Ein effizienterer randomisierter MinCut-Algorithmus . . . 121
8.3 Grundlagen der Wahrscheinlichkeitstheorie II . . . 122
8.4 Das Maximum Satisfiability Problem . . . 123
8.4.1 Der Algorithmus Random Sat . . . 124
8.5 Das MaxCut-Problem . . . 124
8.5.1 Ein Randomisierter Algorithmus für MaxCut . . . 124
8.5.2 Relaxierung von IQP . . . 125

**9 Parallele Algorithmen 129**
9.1 Das PRAM Modell . . . 130
9.2 Komplexität von parallelen Algorithmen . . . 130
9.3 Die Komplexitätsklassen . . . 131
9.4 Parallele Basisalgorithmen . . . 132
9.4.1 Broadcast . . . 132
9.4.2 Berechnung von Summen . . . 132
9.4.3 Berechnung von Präfixsummen . . . 133
9.4.4 List Ranking . . . 135

9.4.5 Binäroperationen einer partitionierten Menge mit $K$ Prozessoren . . . . . 135
9.5 Zusammenhangskomponenten . . . . . 136
9.6 Modifikation zur Bestimmung eines MST . . . . . 138
9.6.1 Der Algorithmus . . . . . 139
9.6.2 Reduzierung der Prozessorenzahl auf $\frac{n^2}{\log^2 n}$ . . . . . 141
9.7 ParallelSelect . . . . . 142
9.8 Das Scheduling-Problem . . . . . 146

**10 Parametrisierte Algorithmen** **149**
10.1 Parametrisierte Komplexitätstheorie (Exkurs) . . . . . 151
10.2 Grundtechniken zur Entwicklung parametrisierter Algorithmen . . . . . 151
10.3 Kernbildung mit Linearer Programmierung . . . . . 155

# Abbildungsverzeichnis


1 Sortieren mit der konvexen Hülle . . . . . 5

1.1 Mengen bei Union-Find . . . . . 20
1.2 Vereinigung von Mengen bei Union-Find . . . . . 21
1.3 Tiefe eines Baumes . . . . . 22
1.4 „weighted Union“: Union$(i, j)$ . . . . . 23
1.5 Pfadkompression bei Find . . . . . 25
1.6 Rang eines Knoten . . . . . 26
1.7 Deterministischer endlicher Automat $\mathcal{A}$ . . . . . 31
1.8 Deterministischer endlicher Automat $\mathcal{A}_1$ . . . . . 31
1.9 Deterministischer endlicher Automat $\mathcal{A}_2$ . . . . . 32
Beispiel eines Heap . . . . . 35
1.10 Beispielhafte Arbeitsweise von Heapify . . . . . 35

2.1 Ein aufspannender Baum minimalen Gewichts . . . . . 42
2.2 Beispiel eines Schnitts $(S, V \setminus S)$ . . . . . 43
2.3 Ein Kreis in einem Graphen . . . . . 43
2.4 Eine Anwendung der grünen Regel . . . . . 44
2.5 Eine Anwendung der roten Regel . . . . . 44
2.6 Die Invariante der grünen Regel . . . . . 45
2.7 Unabhängige Knotenmengen sind kein Matroid . . . . . 50

3.1 Beispiel eines minimalen Schnitts . . . . . 53
3.2 Ein Beispiel für das Verschmelzen von Knoten . . . . . 54
1. Phase . . . . . 55
2. Phase . . . . . 55
3. Phase . . . . . 56
4. Phase . . . . . 56
5. Phase . . . . . 56
6. Phase . . . . . 57
7. Phase . . . . . 57

Erster aktiver Knoten $v \notin S'$ . . . 59
Erster aktiver Knoten $v \in S'$ . . . 59
Nächster aktiver Knoten . . . 59

Ein Fluss und Schnitte . . . 65
Negativbeispiel für Ford-Fulkerson . . . 68
Edmonds und Karp beim Start . . . 69
Edmonds und Karp nach Update . . . 69
4.1 Flussnetzwerk mit Präfluss . . . 73
Beispiel zu Goldberg-Tarjan . . . 75
4.2 Flussnetzwerk für Mehrmaschinen-Scheduling . . . 82

Beispiel einer Kreisbasis, die aus einem Spannbaum gewonnen wird. . . . 84
5.1 Elemente der Basis können stets durch ihren Schnitt mit einem anderen Basiselement ersetzt werden . . . 85
5.2 Elemente der Basis können stets verkleinert werden . . . 86
5.3 Darstellung eines Kreises der MCB durch kürzeste Wege . . . 87
5.4 Ein Graph zur Illustration des Algorithmus von de Pina . . . 88

6.1 Geometrische Darstellung eines LP . . . 96
6.2 Klee-Minty-Polyeder . . . 97

Worst-case-Beispiel zu Next Fit . . . 102
Worst-case-Beispiel zu First Fit . . . 104
Jobzuteilung beim Multiprocessor Scheduling . . . 107

8.1 Eine Lösung von $\mathrm{QP}^2(I)$. . . . 126
8.2 Runden der zweidimensionalen Lösung . . . 127

10.1 Independent Set, Vertex Cover und Domintating Set eines Beispiel-Graphen. . . . 150

# Algorithmenverzeichnis


1 FINDE MAXIMUM . . . 1
2 QUICKSORT . . . 3
3 MULTIPOP$(S, k)$ . . . 6
4 Iteratives MERGE-SORT$(S, k)$ . . . 9
5 Rekursives MERGE-SORT$(A, p, r)$ . . . 9
6 SLOWSELECT$(A, i)$ . . . 16
7 SELECT$(A, i)$ . . . 16
8 MAKESET$(x)$ (#1) . . . 20
9 FIND$(x)$ (#1) . . . 20
10 UNION$(S_i, S_j, S_k)$ (#1) . . . 20
11 FIND$(x)$ (#2) . . . 21
12 UNION$(i, j)$ (#2) . . . 21
13 MAKESET$(x)$ (#2) . . . 21
14 UNION$(i, j)$ (#3) . . . 22
15 FIND$(x)$ (#3) . . . 24
16 Algorithmus von Kruskal . . . 29
17 OFFLINE-MIN . . . 30
18 Äquivalenz endlicher Automaten . . . 33
19 HEAPIFY$(A, i)$ . . . 36
20 MAKEHEAP $(M)$ . . . 36
21 DELETE $(A, i)$ . . . 37
22 SIFT-UP$(A, i)$ . . . 38
23 INSERT$(A, x)$ . . . 38
24 HEAPSORT$(A)$ . . . 39
25 BOTTOM-UP-HEAPIFY$(A, 1)$ . . . 40
26 Färbungsmethode von Tarjan . . . 44
27 Algorithmus von Kruskal (verbal) . . . 46
28 Algorithmus von Prim (verbal) . . . 47
29 Algorithmus von Prim . . . 48
30 Greedy–Methode für ein Optimierungsproblem . . . 49
31 MINSCHNITTPHASE$(G_i, c, a)$ . . . 58
32 MIN–SCHNITT$(G, c, a)$ . . . 58
33 MAX-FLOW . . . 66
34 Algorithmus von Ford-Fulkerson . . . 67
35 Algorithmus von Goldberg-Tarjan . . . 74
36 Prozedur PUSH . . . 74
37 Prozedur RELABEL . . . 74
38 MCB-GREEDY-METHODE . . . 85
39 Algorithmus von Horton . . . 87
40 Algorithmus von de Pina ([3]) . . . 87
41 Algorithmus von de Pina algebraisch . . . 90
42 SIMPLE MCB . . . 90

43 MCB-Checker . . . 91
44 Greedy-Knapsack . . . 101
45 Next Fit (NF) . . . 102
46 First Fit (FF) . . . 103
47 List Scheduling . . . 106
48 Algorithmus $\mathcal{A}_\ell$ für Multiprocessor Scheduling . . . 107
49 APAS für Bin Packing . . . 113
50 Random MinCut . . . 119
51 Fast Random MinCut . . . 121
52 Random MaxCut . . . 125
53 Broadcast($N$) . . . 132
54 Summe($a_1, \ldots, a_n$) . . . 132
55 Oder($x_1, \ldots, x_n$) . . . 133
56 Präfixsumme($a_n, \ldots, a_{2n-1}$) . . . 134
57 List Ranking($n$, $h$) . . . 135
58 Zusammenhang($G$) [Chandra, Hirschberg & Sarwarte 1979] . . . 137
59 MST($G$) . . . 139
60 ParallelSelect($S, k; p$) . . . 144
61 ParallelScheduling . . . 147
62 Vertex-Cover($G, k$) . . . 152

”When I consider what people generally want in calculating, I found that it always is a number. I also observed that every number is composed of units, and that any number may be divided into units. Moreover, I found that every number which may be expressed from one to ten, surpasses the preceding by one unit: afterwards the ten is doubled or tripled just as before the units were: thus arise twenty, thirty, etc. until a hundred: then the hundred is doubled and tripled in the same manner as the units and the tens, up to a thousand; ... so forth to the utmost limit of numeration.”


Abu Ja’far Muhammad ibn Musa Al-Khwarizmi $\sim$ 820
(Namensgeber des Begriffs „Algorithmus“)

# Kapitel 0

# Grundlagen

**Der Literaturtip.** Ein gutes Buch unter vielen zum Gesamtthema der Vorlesung ist [2]. Es enthält auch einführende Kapitel zum Algorithmusbegriff sowie zur Laufzeitanalyse von Algorithmen und zu Wachstumsraten von Funktionen $(O, \Omega, \Theta)$. Ein weiteres empfehlenswertes Buch ist [17].

## 0.1 Worst-case- und Average-case-Laufzeit

**Definition 0.1 (Worst-case-Laufzeit).** *Bezeichne die Anzahl Operationen, die ein Algorithmus $\mathcal{A}$ bei Eingabe $x$ ausführt, mit $t_{\mathcal{A}}(x)$. Die Worst-case-Laufzeit eines Algorithmus $\mathcal{A}$ ist:*

$$T_{\mathcal{A}}^{\text{worst}}(n) := \max_{x \text{ Eingabe}, |x|=n} t_{\mathcal{A}}(x)$$

**Definition 0.2 (Average-case-Laufzeit).** *Die Average-case-Laufzeit eines Algorithmus $\mathcal{A}$ ist:*

$$T_{\mathcal{A}}^{\text{av}}(n) := \frac{1}{|\{x : x \textit{ ist Eingabe mit } |x| = n\}|} \sum_{x:|x|=n} t_{\mathcal{A}}(x)$$

Dies bedeutet, $T_{\mathcal{A}}^{\text{av}}(n)$ gibt die „mittlere“ Laufzeit bzw. den *Erwartungswert* für die Laufzeit bei Eingabe der Länge $n$ an, wobei *Gleichverteilung* unter allen Eingaben angenommen wird. Somit gilt für die Zufallsvariable $t_{\mathcal{A}}(x)$: $T_{\mathcal{A}}^{\text{av}}(n) = E[t_{\mathcal{A}}(x)]$.

### 0.1.1 Beispiel: Worst-case-Laufzeit von Finde Maximum

Es sei $A[1, \ldots, n], n \geq 1$ ein Array der Länge $n$. Bestimme das Maximum der Werte $A[1], \ldots, A[n]$.

**Algorithmus 1** : FINDE MAXIMUM

**Eingabe** : Unsortiertes Array von $n$ verschiedenen Zahlen $A = (A[1], \ldots, A[n])$
**Ausgabe** : Maximum aller $A[i]$

```
1 max ← A[1]
2 for i ← 2 to n do
3 |   Wenn A[i] > max
4 |   |   max ← A[i]
```

$c_1$ Aufwand von Schritt **1**

$c_2$ Aufwand von Schritt **2** und **3** bei einem Durchlauf

$c_3$ Aufwand von Schritt **4**

Also: $T_{\mathcal{A}}^{\text{worst}}(n) \leq c_1 + (n-1)(c_2 + c_3)$

### 0.1.2 Beispiel: Average-case-Laufzeit von Finde Maximum

Es sei $A[1, \ldots, n]$ eine beliebige Permutation der Zahlen $A[1], \ldots, A[n]$. Alle $n!$ Permutationen seien gleich wahrscheinlich. Somit gilt:

$$\begin{aligned}
&\Pr(A[2] > A[1]) && = \frac{1}{2} \\
&\Pr(A[3] > \max(A[1], A[2])) && = \frac{1}{3} \\
&\quad \vdots \\
&\Pr(A[n] > \max(A[1], \ldots, A[n-1])) && = \frac{1}{n}
\end{aligned}$$

Nun sei $X_j$ für $j \in \{1, \ldots, n\}$ eine Zufallsvariable mit:

$$X_j = \begin{cases} 1 & \text{falls } A[j] \text{ Maximum in } A[1, \ldots, j] \\ 0 & \text{sonst} \end{cases} \tag{1}$$

dann ist der Erwartungswert

$$\begin{aligned}
E[X_j] &= \Pr(A[j] > max(A[1], \ldots, A[j-1])) \\
&= \frac{1}{j}.
\end{aligned}$$

Für Algorithmus 1 gilt somit:

$$\begin{aligned}
T_{\mathcal{A}}^{\text{av}}(n) &= c_1 + (n-1)c_2 + c_3 E\left[\sum_{j=2}^{n} X_j\right] \\
&= c_1 + (n-1)c_2 + c_3 \sum_{j=2}^{n} E[X_j] \\
&= c_1 + (n-1)c_2 + c_3 \underbrace{\sum_{j=2}^{n} \frac{1}{j}}_{H_n - 1}
\end{aligned}$$

Hier ist $H_n$ die harmonische Reihe, für die gilt: $\ln(n+1) \leq H_n \leq \ln n + 1$. Somit gilt:

$$T_{\mathcal{A}}^{\text{av}}(n) \leq c_1 + (n-1)c_2 + c_3 \ln n$$

**Algorithmus 2** : QUICKSORT

**Eingabe** : Unsortiertes Array von $r - l$ verschiedenen Zahlen $A = (A[l], \ldots, A[r])$
**Ausgabe** : Sortiertes Array $A$

```
1 Wenn r − l > 0
2     x ← A[l]
3     Ordne A = (A[l, ..., r]) so um, dass alle Elemente, die kleiner bzw. größer als x sind,
      links, respektive rechts von x in A stehen.
4     Sei das Element x danach an Position q in A
5     QUICKSORT A[l, ..., q − 1]
6     QUICKSORT A[q + 1, ..., r]
```

### 0.1.3 Beispiel: Average-case-Laufzeit von Quicksort

Wir wollen die Average-case-Laufzeit von QUICKSORT (Algorithmus 2) bestimmen. Im Folgenden zählen wir für die Laufzeit nur noch die Anzahl der Vergleiche, die ein Sortieralgorithmus braucht.

**Bemerkung 0.3.** *Der Worst-case tritt ein, falls zum Beispiel $A$ bereits vorsortiert ist. Dann werden im ersten Aufruf $n-1$ Vergleiche, im nächsten $n-2$ Vergleiche etc. gemacht. Somit folgt:* $T^{\text{worst}}_{\text{QUICKSORT}}(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$*, wobei wir in $T$ nur die Vergleiche zählen.*

Annahme: Alle $n!$ Permutationen sind gleich wahrscheinlich. Es bezeichne $s_i$ das $i$-kleinste Element in $A$. Weiterhin seien $X_{ij}$ Zufallsvariablen, definiert für alle Paare $i < j$:

$$X_{i,j} = \begin{cases} 1 & \text{falls } s_i \text{ und } s_j \text{ verglichen werden} \\ 0 & \text{sonst} \end{cases} \tag{2}$$

**Bemerkung 0.4.** *Die Elemente $s_i$ und $s_j$ werden höchstens einmal verglichen.*

Es gilt:

$$T^{\text{av}}_{\text{QUICKSORT}}(n) = E\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} E[X_{ij}]$$

und

$$E[X_{ij}] = 1 \cdot p_{ij} - 0 \cdot (1 - p_{ij}) = p_{ij}$$

wobei $p_{ij}$ die Wahrscheinlichkeit ist, dass $s_i$ und $s_j$ verglichen werden.

**Beobachtung 0.5.** *Die Elemente $s_i$ und $s_j$ werden genau dann verglichen, wenn $\mu = i$ oder $\mu = j$, wobei $s_\mu = A[k]$, $k$ linkeste Stelle in $A$, an der ein Element aus $s_i, s_{i+1}, \ldots, s_j$ steht.*

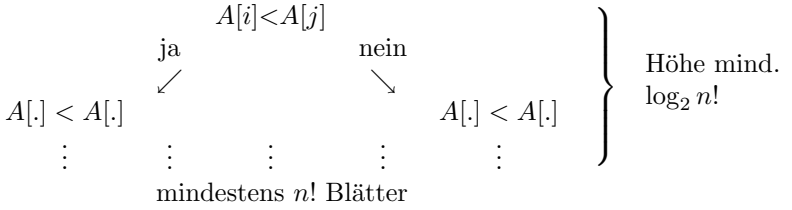
Somit folgt: $p_{ij} = \frac{2}{j-i+1}$ und

$$\begin{aligned}
T^{\text{av}}_{\text{QUICKSORT}}(n) &= \sum_{i<j} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1+(1-i)}^{n+(1-i)} \frac{2}{j-i+1-(1-i)} = 2 \cdot \sum_{i=1}^{n-1} \sum_{j=2}^{n+1-i} \frac{1}{j} \\
&\leq 2 \cdot \sum_{i=1}^{n-1} \underbrace{\sum_{j=2}^{n} \frac{1}{j}}_{H_n - 1} \quad \text{harmonische Reihe!} \\
&\leq 2n(H_n - 1) \leq 2n \ln n \\
&(\leq 1{,}3863 n \log_2 n)
\end{aligned}$$

## 0.2 Untere Schranken für Laufzeiten

### 0.2.1 „Sortieren ist in $\Omega(n \log n)$“

Eingabe eines Sortieralgorithmus ist ein Array $A = A[1, \ldots, n]$. Die Ausgabe ist das Array in sortierter Form und somit eine von $n!$ möglichen Permutationen von $A[1, \ldots, n]$. Ein beliebiges Sortierverfahren, welches eine Abfolge von Vergleichen ausführt, entspricht einem *Entscheidungsbaum* wie folgt:




Somit gilt für jeden Sortieralgorithmus, der auf Vergleichen von je zwei Elementen beruht

$$T^{\text{worst}}_{\text{SORT}}(n) \geq \log_2 n! \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}(\log_2 n - 1) \in \Omega(n \log n)$$

Algorithmen, die nicht nur auf dem Vergleichen von zwei Elementen beruhen, fordern in der Regel spezielle Voraussetzungen, wie zum Beispiel BUCKET SORT, oder erlauben spezielle algebraische Operationen. Gegebenenfalls ist Sortieren dann in Linearzeit möglich. Es gilt übrigens auch: $T^{\text{av}}_{\text{SORT}}(n) \in \Omega(n \log n)$, da auch die mittlere Blatttiefe $\log n!$ ist.

### 0.2.2 Untere Schranke für Berechnung der konvexen Hülle von $n$ Punkten ist $\Omega(n \log n)$

Berechnung der konvexen Hülle $H(P)$ der Menge $P$ von $n$ Punkten im $\mathbb{R}^2$, d.h. das minimale konvexe Polygon, so dass alle Punkte aus $Q$ im Innern oder auf dem Rand des Polygons liegen.

- *Graham-Scan-Algorithmus* in $O(n \log n)$
- *Jarvis-March-Algorithmus* in $O(nh)$, mit $h$ Anzahl der Punkte auf dem Rand von $H(P)$

Annahme: Es gibt einen Algorithmus $\mathcal{A}$, der $H(P)$ in $T_{\mathcal{A}}(n) \in o(n \log n)$ berechnet. Konstruiere aus $n$ Zahlen $a_1, \ldots, a_n$ eine Punktemenge $P := \{(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_n, a_n^2)\}$. Die Sortierung der Zahlen $a_1, \ldots, a_n$ ist dann möglich mittels Berechnung von $H(P)$ und Minimum $a_{\min}$ der Zahlen $a_1, \ldots, a_n$ durch Ablauf des Randes von $H(P)$ im Gegenuhrzeigersinn ab $(a_{\min}, a_{\min}^2)$, denn alle Punkte aus $P$ liegen auf dem Rand von $H(P)$. Siehe dazu Abbildung 1.
Also: $T_{\text{SORT}}(n) \in O(T_{\mathcal{A}}(n) + n)$, ein Widerspruch zu „Sortieren ist in $\Omega(n \log n)$“ (da wir auch hier wie üblich lediglich Vergleiche zulassen und i.A. keine weiteren Voraussetzungen haben).

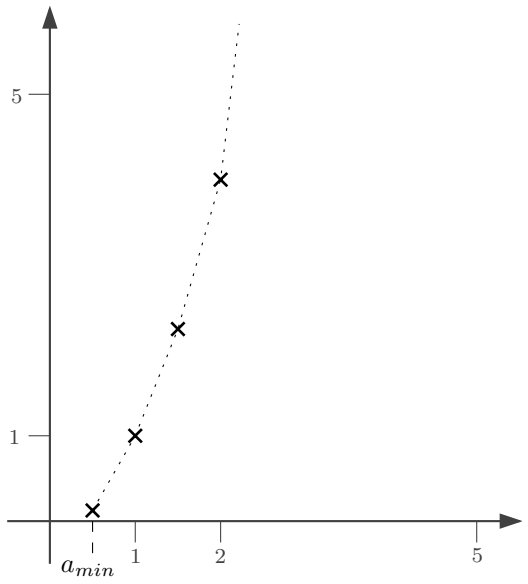



Abbildung 1: Die konvexe Hülle dieser Punktemenge liefert eine Sortierung

## 0.3 Amortisierte Analyse

Die *amortisierte Laufzeitanalyse* ist eine Analysemethode, bei der die Zeit, die benötigt wird, um eine Folge von Operationen auszuführen, über alle auszuführenden Operationen gemittelt wird. Diese Analysemethode kann angewandt werden, um zu beweisen, dass die mittleren Kosten einer Operation klein sind, wenn man über eine Folge von Operationen mittelt, obwohl eine einzelne Operation sehr aufwendig sein kann. Im Unterschied zu der „Average-case“-Analyse werden keine Wahrscheinlichkeitsannahmen gemacht. Analyse-Techniken für die amortisierte Analyse sind:

1. **Ganzheitsmethode:** Bestimme eine obere Schranke $T(n)$ für die Gesamtkosten von $n$ Operationen. Das ergibt $\frac{T(n)}{n}$ als die amortisierten Kosten für eine Operation.

2. **Buchungsmethode (engl. accounting):** Bestimme die amortisierten Kosten jeder Operation. Verschiedene Operationen können verschiedene Kosten haben. Es werden frühe Operationen in der betrachteten Folge höher veranschlagt und die zu hoch veranschlagten Kosten jeweils als Kredit für festgelegte Objekte gespeichert. Dieser Kredit wird dann für spätere Operationen verbraucht, deren Kosten niedriger veranschlagt werden, als sie tatsächlich sind.

3. **Potentialmethode:** Ähnlich wie bei der Buchungsmethode werden die amortisierten Kosten jeder einzelnen Operation berechnet. Dabei werden möglicherweise wieder einzelne Operationen höher veranschlagt und später andere Operationen niedriger veranschlagt. Der Kredit wird als Potentialenergie insgesamt gespeichert, anstatt einzelnen Objekten zugeordnet zu werden.

Beachte dabei: Der Kredit wird nur für die Analyse eingeführt.

### 0.3.1 Beispiel von Stackoperationen

Wir wollen die drei Methoden am Beispiel „STACK mit MULTIPOP“ erläutern. Betrachte dazu eine Datenstruktur mit „Last in – First out“, einen STACK $S$. Ein „normaler“ STACK umfasst zwei Operationen:

PUSH$(S, x)$: lege Objekt $x$ auf STACK $S$

POP$(S)$: gib oberstes Objekt vom STACK aus und entferne es vom STACK.

Beide Operationen sind in $\Theta(1)$. Veranschlage daher die Kosten pro Operation mit 1. Dann sind die Gesamtkosten einer Folge von insgesamt $n$ PUSH- und POP-Operationen in $\Theta(n)$. Betrachte nun eine weitere Operation:

MULTIPOP$(S, k)$: gib die obersten $k$ Objekte vom STACK aus und entferne sie vom STACK, bzw. falls der STACK weniger als $k$ Objekte enthält, gib alle Objekte aus, bis der STACK leer ist.

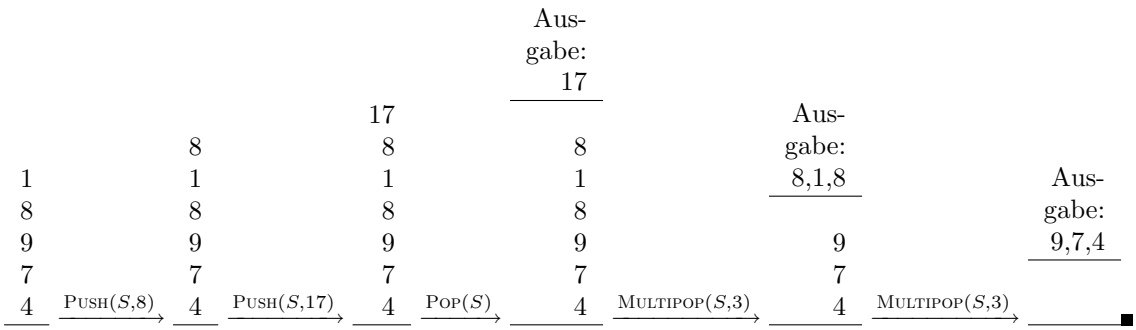
Eine formale Beschreibung von MULTIPOP$(S, k)$ zeigt Algorithmus 3.

**Algorithmus 3** : MULTIPOP$(S, k)$

**Eingabe** : STACK, Anzahl zu entfernender Elemente
**Ausgabe** : Die $k$ obersten Elemente des STACK
**Seiteneffekte** : STACK verkleinert sich um bis zu $k$ Elemente

```
1 Solange S ≠ ∅ und k ≠ 0 tue
2 |  POP(S)
3 |  k ← k − 1
```

**Beispiel 0.6.**




Die Laufzeit einer Operation MULTIPOP$(S, k)$ ist linear in der Anzahl der ausgeführten POP-Operationen, also $O(\min(s, k))$, wobei $s = |S|$ ist und wobei die Kosten im Worst-case betrachtet werden. Für eine Folge von $n$ PUSH-, POP- und MULTIPOP-Operationen, angewandt auf einen zu Beginn leeren STACK, ergibt sich:

- Worst-case-Laufzeit einer MULTIPOP-Operation ist $O(n)$
- Worst-case-Laufzeit einer beliebigen STACK-Operation ist $O(n)$
- Für $n$ STACK-Operationen ergibt sich dann im Worst-case $O(n^2)$.

### 0.3.2 Die Ganzheitsmethode

Eine einzelne MULTIPOP-Operation kann teuer sein, d.h. sie kann Kosten $n$ haben. Andererseits gilt: Ein Objekt kann höchstens einmal wieder einen POP erfahren für jedes Mal, dass es einen PUSH erfahren hat. Bei dieser Rechnung sind die POP-Operationen von MULTIPOP inbegriffen. Es sind also höchstens so viele Kosten für POP-Operationen möglich wie für PUSH-Operationen, d.h. höchstens $n$. Die Gesamtlaufzeit ist also in $O(n)$, und damit ist der amortisierte Aufwand einer einzelnen Operation in der Folge der $n$ STACK-Operationen in $O(1)$.

### 0.3.3 Die Buchungsmethode

Es werden verschiedene „Gebühren“ für verschiedene Operationen veranschlagt. Die Gebühr pro Operation ist dann der amortisierte Aufwand der Operation. Falls der amortisierte Aufwand einer einzelnen Operation höher ist als der wirkliche Aufwand, so wird die Differenz als Kredit speziellen Objekten zugeordnet. Dieser Kredit kann später benutzt werden, um Aufwand, der höher als der amortisierte Aufwand einer Operation ist, auszugleichen. Der amortisierte Aufwand muss sorgfältig veranschlagt werden. Der Kredit darf niemals negativ sein, denn die gesamte amortisierte Laufzeit einer Folge von Operationen muss immer eine obere Schranke für die wirkliche Laufzeit sein.

| | | |
|---|---|---|
| Tatsächlicher Aufwand bei STACK-Operationen: | PUSH$(S, x)$: | 1 |
| | POP$(S)$: | 1 |
| | MULTIPOP$(S, k)$: | $\min(\lvert S\rvert, k)$ |
| Definiere amortisierten Aufwand (Gebühr): | PUSH$(S, x)$: | 2 |
| | POP$(S)$: | 0 |
| | MULTIPOP$(S, k)$: | 0 |

**Beispiel 0.7 (Tabletts in der Mensa).** Wenn wir ein Tablett auf den STACK legen, bezahlen wir zwei Einheiten. Eine Einheit wird sofort verwendet, um die wirklichen Kosten der PUSH-Operation zu bezahlen. Die zweite Einheit bleibt auf dem Tablett liegen. Wenn das Tablett ein POP-Operation erfährt, egal ob durch POP oder MULTIPOP, wird die Einheit auf dem Tablett verwendet, um die wirklichen Kosten für POP zu bezahlen. Offensichtlich ist der Kredit niemals negativ. Bei einer Folge von $n$ PUSH-, POP- und MULTIPOP-Operationen benötigen wir höchstens $2 \cdot n$ amortisierte Gesamtkosten, da für maximal $n$ PUSH-Operationen je zwei als amortisierte Kosten veranschlagt werden. Die Gesamtlaufzeit ist also in $O(n)$. ∎

### 0.3.4 Die Potentialmethode

Es wird aus zu hoch veranschlagten Kosten ein Potential aufgespart, das später verbraucht werden kann. Starte mit einer Datenstruktur $D_0$, auf der $n$ Operationen ausgeführt werden.

Für $i = 1, 2, \ldots, n$ seien
- $c_i$: die tatsächlichen Kosten der $i$-ten Operation
- $D_i$: die Datenstruktur nach der $i$-ten Operation (angewandt auf $D_{i-1}$).

Definiere eine Potentialfunktion $\mathbb{C} : D_i \mapsto \mathbb{C}(D_i)$ „Potential von $D_i$“ und den amortisierten Aufwand $\widehat{c}_i$ der $i$-ten Operation bzgl. $\mathbb{C}$ als

$$\widehat{c}_i := c_i + \mathbb{C}(D_i) - \mathbb{C}(D_{i-1})$$

Die amortisierten Kosten sind also die tatsächlichen Kosten plus dem Zuwachs (oder Verlust) an

Potential entsprechend der Operation. Das heißt:

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} (c_i + \mathbb{C}(D_i) - \mathbb{C}(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \mathbb{C}(D_n) - \mathbb{C}(D_0).\end{aligned}$$

Wenn $\mathbb{C}$ so definiert wird, dass $\mathbb{C}(D_n) \geq \mathbb{C}(D_0)$ ist, dann sind die amortisierten Kosten eine obere Schranke für die Gesamtkosten. Im allgemeinen ist nicht unbedingt im vorhinein klar, wieviele Operationen ausgeführt werden. Falls $\mathbb{C}(D_i) \geq \mathbb{C}(D_0)$ für alle $i$, so ist allerdings garantiert, dass (wie bei der Buchungsmethode) im voraus immer genug Kredit angespart wurde. Oft ist es günstig, $\mathbb{C}$ so zu wählen, dass $\mathbb{C}(D_0) = 0$ ist und zu zeigen, dass $\mathbb{C}(D_i) \geq 0$ ist für alle $i$.

Intuitiv ist klar: Falls $\mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) > 0$ ist, dann wird durch die amortisierten Kosten $\widehat{c}_i$ eine überhöhte Gebühr für die $i$-te Operation dargestellt, d.h. dass das Potential wächst. Falls $\mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) < 0$ ist, so ist $\widehat{c}_i$ zu tief veranschlagt.

**Beispiel:** STACK **mit** MULTIPOP**:** Definiere $\mathbb{C}$ als Anzahl der Objekte auf dem STACK. Dann ist $\mathbb{C}(D_0) = 0$. Da die Anzahl der Objekte auf einem STACK nie negativ ist, gilt für alle $i$ $\mathbb{C}(D_i) \geq \mathbb{C}(D_0)$. $\mathbb{C}$ ist also gut gewählt, da die amortisierten Gesamtkosten $\sum_{i=1}^{\ell} c_i + \mathbb{C}(D_\ell) - \mathbb{C}(D_0)$ zu jedem Zeitpunkt $\ell$ eine obere Schranke für die wirklichen Gesamtkosten sind. Betrachte den amortisierten Aufwand für die verschiedenen STACK-Operationen:

1. Die $i$-te Operation sei ein PUSH, auf einen STACK mit $s$ Objekten angewandt:

   $$c_i = 1 \text{ und } \mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) = (s+1) - s = 1.$$

   Daraus folgt

   $$\widehat{c}_i = c_i + \mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) = 1 + 1 = 2.$$

2. Die $i$-te Operation sei ein POP, angewandt auf einen STACK mit $s$ Objekten:

   $$c_i = 1 \text{ und } \mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) = s - 1 - s = -1.$$

   Daraus folgt

   $$\widehat{c}_i = 1 - 1 = 0.$$

3. Die $i$-te Operation sei ein MULTIPOP$(S, k)$, auf einen STACK $S$ mit $s$ Objekten und $k' := \min(|S|, k)$ angewandt:

   $$c_i = k' \text{ und } \mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) = (s - k') - s = -k'.$$

   Daraus folgt

   $$\widehat{c}_i = c_i + \mathbb{C}(D_i) - \mathbb{C}(D_{i-1}) = k' - k' = 0.$$

Also ist $\sum_{i=1}^{n} \widehat{c}_i \leq 2n$; die amortisierte Gesamtlaufzeit ist damit im Worst-case in $O(n)$.

**Bemerkung 0.8.** *Ein weiteres Beispiel, an dem sich die Ideen der amortisierten Analyse gut erklären lassen, sind* dynamische Tabellen*: Objekte sollen in einer Tabelle abgespeichert werden, wobei Objekte eingefügt bzw. gelöscht werden können. Zu Beginn einer Folge von Operationen vom Typ Einfügen und Löschen habe die Tabelle die Größe $h$. Wenn eine Operation Einfügen zu einem Zeitpunkt auf die Tabelle angewandt wird, an dem die Tabelle voll ist, soll eine Tabellenexpansion vorgenommen werden, in der die Tabelle verdoppelt wird. Die Tabellenexpansion (angewandt auf die Tabelle der Größe $k$) habe wirkliche Kosten $k$. Entsprechend werde eine Tabellenkontraktion durchgeführt, in der die Tabelle halbiert wird, wenn sie nur noch sehr dünn besetzt ist, etwa höchstens zu $\frac{1}{4}$. Die Tabellenkontraktion angewandt auf eine Tabelle mit $k$ Elementen hat dann wiederum Kosten $k$. Wie groß ist die amortisierte Laufzeit für eine Folge von $n$ Operationen vom Typ Einfügen bzw. Löschen?*

## 0.4 Divide-and-Conquer Verfahren und das Prinzip der Rekursion

Ein grundlegendes algorithmisches Prinzip besteht darin, ein Problem zu lösen, indem man es in Probleme (meist desselben Typs) kleinerer Größe oder mit kleineren Werten aufteilt, diese löst und aus den Lösungen eine Lösung für das Ausgangsproblem konstruiert. Wir werden in den angegebenen rekursiven Algorithmen oft keine explizite Abbruchbedingung angeben. Es wird angenommen, dass eine geeignete Abbruchbedingung eingebaut wird.

### 0.4.1 Ein Beispiel aus der Informatik-Grundvorlesung

MERGE-SORT ist ein Verfahren zum Sortieren einer Folge von $n$ Werten, auf denen es eine Ordnung „$\leq$“ gibt. MERGE-SORT sortiert eine Folge der Länge $n$, indem es zunächst halb so lange Teilfolgen sortiert und aus diesen die sortierte Folge der Länge $n$ „zusammmenmischt“. Dies kann, wie im Folgenden beschrieben, auf iterative oder auf rekursive Weise durchgeführt werden.

#### Iterativ bzw. Bottom-up

**Algorithmus 4** : Iteratives MERGE-SORT$(S, k)$

1 Sortiere zunächst Teilfolgen der Länge zwei und mische sie zu sortierten Folgen der Länge vier zusammen
2 Mische je zwei sortierte Folgen der Länge vier zu sortierten Folgen der Länge acht zusammen
3 u.s.w.

#### Rekursiv

Sehr oft werden Divide-and-Conquer Verfahren jedoch rekursiv angewandt. Das bedeutet: Das Verfahren ruft sich selbst wieder auf, angewandt auf Eingaben kleinerer Länge bzw. mit kleineren Werten.

**Algorithmus 5** : Rekursives MERGE-SORT$(A, p, r)$

**Eingabe** : Array $A$ mit $n$ Elementen, die an den Stellen $A[p]$ bis $A[r]$ stehen.
**Ausgabe** : $n$ Elemente sortiert in Ergebnisarray $B$.

1 **Wenn** $p < r$ *gilt*
2 $\quad q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
$\quad B_1 \leftarrow$ MERGE-SORT$(A; p, q)$ } rekursive Aufrufe
3 $\quad B_2 \leftarrow$ MERGE-SORT$(A; q+1, r)$ }
4 $\quad B \leftarrow$ MERGE$(B_1; B_2)$
5 **sonst**
6 $\quad B \leftarrow A[p]$

Die Hauptarbeit besteht hier bei MERGE-SORT (vgl. Algorithmus 5), wie auch bei den meisten Divide-and-Conquer Verfahren, im „Zusammensetzen“ der Teillösungen, bei MERGE-SORT also in MERGE (Zeile 4).

MERGE **informell:** Durchlaufe $A$ einerseits von $A[p]$ bis $A[q]$ und andererseits von $A[q+1]$ bis $A[r]$ und vergleiche die Elemente jeweils paarweise. Das kleinere der Elemente wird „weggeschrieben“

(in ein Ergebnisarray) und in dem entsprechenden Teil von $A$ um eine Position weitergegangen. Ist man bei $A[q]$ oder $A[r]$ angelangt, wird der restliche Teil des anderen Teilarrays von $A$ an das Ergebnisarray angehängt. Die Einträge aus dem Ergebnisarray könnten dann an die Stellen $A[p]$ bis $A[q]$ kopiert werden.

MERGE hat also eine Laufzeit, die linear in der Länge der Eingabe ist, also in $O(n)$ ist. Aus der rekursiven Beschreibung für MERGE-SORT können wir als Laufzeit $T(n)$ insgesamt ablesen:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \underbrace{k \cdot n}_{\text{für MERGE}} \quad \text{mit } k \geq 0 \text{ Konstante.}$$

Solche Rekursionsgleichungen sind typisch für rekursiv beschriebene Divide-and-Conquer-Algorithmen. Wie schätzt man eine Laufzeitfunktion, die als Rekursionsgleichung (bzw. Rekursionsungleichung) gegeben ist, möglichst gut ab?

Zur Erinnerung: Bei MERGE-SORT ist $T(n) \in O(n \cdot \log n)$. Der Beweis wird induktiv geführt, vgl. Informatik Grundvorlesungen.

### 0.4.2 Methoden zur Analyse von Rekursionsabschätzungen

Die Laufzeit eines Algorithmus, der nach dem Prinzip 'teile und herrsche' arbeitet ist oftmals leicht abschätzbar in Form einer Rekursionsabschätzung wie beispielsweise:

$$T(n) = \sum_{i=1}^{m} T(\frac{n}{2}) + f(n).$$

**Beispiel 0.9 (Laufzeit von MERGESORT).** Der Algorithmus MERGESORT hat die Laufzeit :

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + cn \qquad (c > 0 \text{ konstant}) \\ &\in \Theta(n \log n) \end{aligned}$$

Siehe dazu Informatik I + II. ∎

Die folgenden Methoden stehen zur Verfügung:

1. **Substitutionsmethode:** Wir vermuten eine Lösung und beweisen deren Korrektheit induktiv.

2. **Iterationsmethode:** Die Rekursionsabschätzung wird in eine Summe umgewandelt und dann mittels Techniken zur Abschätzung von Summen aufgelöst.

3. **Meistermethode:** Man beweist einen allgemeinen Satz zur Abschätzung von rekursiven Ausdrücken der Form

$$T(n) = a \cdot T(n/b) + f(n), \text{ wobei } a \geq 1 \text{ und } b > 1.$$

Normalerweise ist die Laufzeitfunktion eines Algorithmus nur für ganze Zahlen definiert. Entsprechend steht in einer Rekursionsabschätzung eigentlich $T(\lfloor n/b \rfloor)$ oder $T(\lceil n/b \rceil)$. Außerdem haben Laufzeitfunktionen $T(n)$ die Eigenschaft, dass zwar $T(n) \in O(1)$ ist für kleine $n$, allerdings oft mit großer $O$-Konstante. Meistens kann man diese Details jedoch vernachlässigen.

### 0.4.3 Die Substitutionsmethode

Betrachte als Beispiel wieder die Laufzeitfunktion, von MERGE-SORT (siehe Algorithmus 5) von der wir wissen, dass sie in $O(n \log n)$ liegt. Es gilt:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \ .$$

Wir beweisen, dass $T(n) \leq c \cdot n \cdot \log n$ für geeignetes $c > 0$, $c$ konstant ist. Dazu nehmen wir als Induktionsvoraussetzung an, dass die Abschätzung für Werte kleiner $n$ gilt, also insbesondere

$$\begin{aligned} T(\lfloor n/2 \rfloor) &\leq c \cdot \lfloor n/2 \rfloor \cdot \log(\lfloor n/2 \rfloor) \text{ und} \\ T(\lceil n/2 \rceil) &\leq c \cdot \lceil n/2 \rceil \cdot \log(\lceil n/2 \rceil) \ . \end{aligned}$$

Im Induktionsschritt erhalten wir also:

$$\begin{aligned} T(n) &\leq c \cdot \lfloor n/2 \rfloor \cdot \log \lfloor n/2 \rfloor + c \cdot \lceil n/2 \rceil \cdot \log \lceil n/2 \rceil + n \\ &\leq c \cdot \lfloor n/2 \rfloor \cdot (\log n - 1) + c \cdot \lceil n/2 \rceil \cdot \log n + n \\ &= c \cdot n \cdot \log n - c \cdot \lfloor n/2 \rfloor + n \\ &\leq c \cdot n \log n \text{ für } c \geq 3, n \geq 2 \ . \end{aligned}$$

Für den Induktionsanfang muss natürlich noch die Randbedingung bewiesen werden, z.B. für $T(1) = 1$. Dies ist oft problematisch. Es gilt beispielsweise für kein $c > 0$, dass $T(1) = 1 \leq c \cdot 1 \cdot \log 1 = 0$ ist. Da uns bei Laufzeitfunktionen allerdings asymptotische Abschätzungen genügen, d.h. Abschätzung für $n \geq n_0$, können wir auch mit der Randbedingung $T(4)$ starten, d.h.

$$T(4) = 2 \cdot T(2) + 4 = 4 \cdot T(1) + 8 = 12 \leq c \cdot 4 \cdot \log 4 = c \cdot 8 \text{ für } c \geq 2 \ .$$

#### Wie kommt man an eine gute Vermutung?

Es gibt keine allgemeine Regel für die Substitutionsannahme, aber einige heuristische „Kochrezepte“. Lautet die Rekursionsgleichung etwa

$$T(n) = 2 \cdot T(n/2 + 17) + n,$$

so unterscheidet sich die Lösung vermutlich nicht substantiell von der Lösung für obige Rekursion, da die Addition von 17 im Argument von $T$ für hinreichend große $n$ nicht so erheblich sein kann. In der Tat ist hier wieder $T(n) \in O(n \log n)$ und dies kann wieder mit Induktion bewiesen werden.

Manchmal lässt sich zwar die korrekte Lösung leicht vermuten, aber nicht ohne weiteres beweisen. So ist vermutlich

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \in O(n).$$

Versucht man jedoch zu beweisen, dass $T(n) \leq c \cdot n$ für geeignetes $c > 0$ ist, so erhält man im Induktionsschritt

$$\begin{aligned} T(n) &\leq c \cdot \lfloor n/2 \rfloor + c \cdot \lceil n/2 \rceil + 1 \\ &= c \cdot n + 1, \end{aligned}$$

aber kein $c > 0$ erfüllt $T(n) \leq c \cdot n$.

**Trick:** Starte mit der schärferen Vermutung $T(n) \leq c \cdot n - b$ für $c > 0$ und einer geeigneten Konstanten $b \geq 0$. Dann gilt

$$\begin{aligned} T(n) &\leq c \cdot \lfloor n/2 \rfloor - b + c \cdot \lceil n/2 \rceil - b + 1 \\ &\leq c \cdot n - 2b + 1 \\ &\leq c \cdot n - b \text{ für } b \geq 1. \end{aligned}$$

Ein weiterer Trick, der oft funktioniert, ist die *Variablenersetzung*, wie in Beispiel 0.10 demonstriert.

**Beispiel 0.10.**

$$T(n) = 2 \cdot T(\lfloor \sqrt{n} \rfloor) + \log n.$$

Setze $m = \log n$, also $n = 2^m$, dann ergibt sich

$$T(2^m) = 2 \cdot T(\lfloor 2^{m/2} \rfloor) + m \leq 2 \cdot T(2^{m/2}) + m.$$

Setzt man nun $S(m) := T(2^m)$, so gilt

$$S(m) \leq 2 \cdot S(m/2) + m \in O(m \log m).$$

Rückübersetzung von $S(m)$ nach $T(m)$ ergibt dann

$$T(n) = T(2^m) = S(m) \in O(m \log m) = O(\log n \log \log n).$$ ∎

### 0.4.4 Die Iterationsmethode

Eine naheliegende Methode zur Auflösung einer Rekursionsgleichung ist deren iterative Auflösung.

**Beispiel 0.11.**

$$\begin{aligned} T(n) &= 3 \cdot T(\lfloor n/4 \rfloor) + n \\ &= n + 3 \cdot (\lfloor n/4 \rfloor + 3 \cdot T(\lfloor n/16 \rfloor)) \\ &= n + 3 \cdot (\lfloor n/4 \rfloor + 3 \cdot (\lfloor n/16 \rfloor + 3 \cdot T(\lfloor n/64 \rfloor))) \\ &= n + 3 \cdot \lfloor n/4 \rfloor + 9 \cdot \lfloor n/16 \rfloor + 27 \cdot T(\lfloor n/64 \rfloor). \end{aligned}$$ ∎

Wie weit muss man die Rekursion iterativ auflösen, bis man die Randbedingungen erreicht?

Der $i$-te Term ist $3^i \cdot \lfloor n/4^i \rfloor$. Die Iteration erreicht im Argument 1, wenn $\lfloor n/4^i \rfloor \leq 1$, d.h. wenn $i \geq \log_4 n$. Dann ergibt sich

$$\begin{aligned} T(n) &\leq n + 3 \cdot n/4 + 9 \cdot n/16 + 27 \cdot n/64 + \ldots + 3^{\log_4 n} \cdot c_1 \\ &\quad \text{für } c_1 \geq 0 \text{ konstant} \\ &\leq n \cdot \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + c_1 \cdot n^{\log_4 3}, \text{ da } 3^{\log_4 n} = n^{\log_4 3} \\ &= 4n + c_1 \cdot n^{\log_4 3} \in O(n), \text{ da } \log_4 3 < 1. \end{aligned}$$

Die Iterationsmethode führt oft zu aufwendigen (nicht unbedingt trivialen) Rechnungen. Durch iterative Auflösung einer Rekursionsabschätzung kann man jedoch manchmal zu einer guten Vermutung für die Substitutionsmethode gelangen.

### 0.4.5 Der Aufteilungs-Beschleunigungssatz (Meistermethode, Master-Theorem)

**Satz 0.12 (Aufteilungs-Beschleunigungssatz (Master-Theorem), eingeschränkte Form).** *Seien $a \geq 1, b > 1, c_1 > 0$ und $c_2 > 0$ Konstanten und $T(n)$ über nichtnegative ganze Zahlen definiert durch*

$$\begin{aligned} c_1 \leq T(1) &\leq c_2 \text{ und} \\ a \cdot T(n/b) + c_1 \cdot n \leq T(n) &\leq a \cdot T(n/b) + c_2 \cdot n. \end{aligned}$$

*Für $n = b^q$ gilt*

*(i)* $T(n) \in \Theta(n)$, *falls* $b > a$.

*(ii)* $T(n) \in \Theta(n \cdot \log n)$, *falls* $a = b$.

*(iii)* $T(n) \in \Theta(n^{\log_b a})$, *falls* $b < a$.

Satz 0.12 ist eingeschränkt auf den Fall, dass $f(n) \in \Theta(n)$ ist, wir beweisen ihn nur für Potenzen von $b$, d.h. $n = b^q$. Letztere Einschränkung wird aus technischen Gründen vorgenommen; man kann für $n \neq b^q$ die Analyse unter Betrachtung der nächsten Potenzen von $b$, d.h. $n_2 = b^{q+1}$ für $n_1 = b^q < n < n_2$ durchführen. Die Einschränkung auf $f(n) \in \Theta(n)$ vereinfacht den Beweis insofern, dass man nicht ausführlich $f(n)$ gegenüber $a \cdot T(n/b)$ abschätzen muss.

*Beweis.* Durch Induktion über $q$ beweisen wir dass

$$T(n) \leq c_2 \cdot n \cdot \sum_{i=0}^{q} (a/b)^i.$$

Für $q = 0$ ergibt sich $T(1) \leq c_2$. Die Behauptung gelte also für $q > 0$. Betrachte $q + 1$:

$$\begin{aligned} T(b^{q+1}) &\leq a \cdot T(b^q) + c_2 \cdot b^{q+1} \\ &\leq a \cdot \Big( c_2 \cdot b^q \cdot \sum_{i=0}^{q} (a/b)^i \Big) + c_2 \cdot b^{q+1} \\ &= c_2 \cdot b^{q+1} \cdot \Big( \sum_{i=0}^{q} (a/b)^{i+1} + 1 \Big) \\ &= c_2 \cdot b^{q+1} \cdot \Big( \sum_{i=1}^{q+1} (a/b)^{i} + 1 \Big) \\ &= c_2 \cdot b^{q+1} \cdot \sum_{i=0}^{q+1} (a/b)^i. \end{aligned}$$

Analog lässt sich auch folgern

$$T(n) \geq c_1 \cdot n \cdot \sum_{i=0}^{q} (a/b)^i.$$

**Fall $b > a$** Dann ist $a/b < 1$ und es gibt Konstante $k_1, k_2 > 0$, so dass

$$c_1 \cdot n \cdot k_1 \leq T(n) \leq c_2 \cdot n \cdot k_2, \quad \text{d.h. } T(n) \in \Theta(n).$$

**Fall $b = a$**

$$\begin{aligned} T(n) &= T(b^q) \leq c_2 \cdot b^q \cdot \sum_{i=0}^{q} 1^i = c_2 \cdot b^q \cdot (q+1) \text{ und} \\ T(n) &\geq c_1 \cdot b^q \cdot (q+1), \text{ also } T(n) \in \Theta(n \log n). \end{aligned}$$

**Fall $b < a$**

$$\begin{aligned} T(n) &= T(b^q) \leq c_2 \cdot b^q \cdot \sum_{i=0}^{q} (a/b)^i = c_2 \cdot \sum_{i=0}^{q} a^i \cdot b^{q-i} \\ &= c_2 \cdot \sum_{i=0}^{q} a^{q-i} \cdot b^i = c_2 \cdot a^q \cdot \sum_{i=0}^{q} (b/a)^i. \end{aligned}$$

Dann gibt es Konstante $k_1, k_2 > 0$, so dass

$$c_1 \cdot \underbrace{a^{\log_b n}}_{=n^{\log_b a}} \cdot k_1 \leq T(n) \leq c_2 \cdot \underbrace{a^{\log_b n}}_{=n^{\log_b a}} \cdot k_2$$

und damit $T(n) \in \Theta(n^{\log_b a})$. □

**Bemerkung 0.13.** *Gelten in der Voraussetzung von Satz 0.12 nur die Beschränkungen nach oben (unten), so gilt immer noch $T(n) \in O(n)$ (bzw. $T(n) \in \Omega(n)$).*

Allgemeiner als Satz 0.12 gilt der folgende Satz:

**Satz 0.14 (Master-Theorem).** *Seien $a \geq 1$ und $b > 1$ Konstanten, $f(n)$ eine Funktion in $n$ und $T(n)$ über nichtnegative ganze Zahlen definiert durch*

$$T(n) = a \cdot T(n/b) + f(n),$$

*wobei $n/b$ für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ steht.*

*(i)* $T(n) \in \Theta(f(n))$ *falls* $f(n) \in \Omega(n^{\log_b a+\varepsilon})$ *und* $af(\frac{n}{b}) \leq cf(n)$ *für eine Konstante* $c < 1$ *und* $n \geq n_0$,

*(ii)* $T(n) \in \Theta(n^{\log_b a} \log n)$ *falls* $f(n) \in \Theta(n^{\log_b a})$,

*(iii)* $T(n) \in \Theta(n^{\log_b a})$ *falls* $f(n) \in O(n^{\log_b a-\varepsilon})$.

Beachte, dass die Fälle aus Satz 0.12 und 0.14 korrespondieren. Es existieren noch allgemeinere Formulierungen dieses Satzes, wie zum Beispiel die folgende:

**Satz 0.15 (Allgemeinere Form des Master-Theorems).** *Es sei*

$$T(n) = \sum_{i=1}^{m} T(\alpha_i n) + f(n),$$

*wobei gilt* $0 < \alpha_i < 1, m \geq 1$ *und* $f(n) \in \Theta(n^k), k \geq 0$.
*Dann ist*

*i)* $T(n) \in \Theta(n^k)$ *falls* $\sum_{i=1}^{m} \alpha_i^k < 1$,

*ii)* $T(n) \in \Theta(n^k \log n)$ *falls* $\sum_{i=1}^{m} \alpha_i^k = 1$,

*iii)* $T(n) \in \Theta(n^c)$ *falls* $\sum_{i=1}^{m} \alpha_i^k > 1$.

*wobei $c$ bestimmt wird durch*

$$\sum_{i=1}^{m} \alpha_i^c = 1$$

Die Beweise der Sätze 0.12, 0.14 und 0.15 lassen sich jeweils per Induktion führen.

**Beispiel 0.16.** MERGE-SORT

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$$

$$b = 2, \quad a = 2, \quad f(n) \in \Theta(n) = \Theta(n^{\log_2 2})$$

Aus Fall (ii) folgt $T(n) \in \Theta(n \cdot \log n)$. ■

**Beispiel: Matrix-Multiplikation nach Strassen**

Gegeben seien zwei $(n \times n)$-Matrizen $A, B$. Berechne $A \cdot B$ mit möglichst wenigen Operationen. Herkömmlich: $C = A \cdot B = (a_{ij}) \cdot (b_{ij}) = (c_{ij})$ mit $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. Die Laufzeit ist insgesamt in $O(n^3)$.

Ein Divide-and-Conquer Ansatz für die Matrixmultiplikation ist:

$$A \cdot B = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = C,$$

wobei die $a_i, b_i, c_i$ $(n/2 \times n/2)$-Matrizen sind. Dann berechnet sich $C$ durch

$$\left.\begin{array}{l} c_1 = a_1 b_1 + a_2 b_3 \\ c_2 = a_1 b_2 + a_2 b_4 \\ c_3 = a_3 b_1 + a_4 b_3 \\ c_4 = a_3 b_2 + a_4 b_4 \end{array}\right\} \begin{array}{c} \text{8 Matrixmultiplikationen} \\ \text{von } (n/2 \times n/2)\text{-Matrizen und} \\ \text{4 Additionen von } (n/2 \times n/2)\text{-Matrizen} \end{array}$$

Addition zweier $(n \times n)$-Matrizen ist in $\Theta(n^2)$. Damit ergibt sich

$$\begin{array}{rcl} T(n) & = & 8 \cdot T(n/2) + c \cdot n^2, \quad c > 0 \text{ Konstante,} \\ c \cdot n^2 & \in & O(n^{\log_2 8 - \varepsilon}) = O(n^{3-\varepsilon}) \text{ mit } 0 < \varepsilon = 1 \\ \Longrightarrow T(n) & \in & \Theta(n^{\log_2 8}) = \Theta(n^3) \quad \text{(leider!)} \end{array}$$

Für die Laufzeit ist offensichtlich die Anzahl der Multiplikationen „verantwortlich“. Das heißt: Wenn man die Anzahl der Multiplikationen der $(n/2 \times n/2)$-Matrizen auf weniger als 8 reduzieren könnte bei ordnungsmäßig gleicher Anzahl von Additionen erhält man eine bessere Laufzeit.

Es gibt einen Divide-and-Conquer Ansatz von Strassen, der nur 7 Multiplikationen bei 18 Additionen verwendet. Als Laufzeit ergibt sich dann

$$\begin{array}{rcl} T(n) & = & 7 \cdot T(n/2) + c \cdot n^2 \\ \Longrightarrow T(n) & \in & \Theta(n^{\log_2 7}). \end{array}$$

Schema:

$$\left.\begin{array}{l} c_1 = P_5 + P_4 - P_2 + P_6 \\ c_2 = P_1 + P_2 \\ c_3 = P_3 + P_4 \\ c_4 = P_5 + P_1 - P_3 - P_7 \end{array}\right\} \text{ mit } \left\{\begin{array}{l} P_1 = a_1 \cdot (b_2 - b_4) \\ P_2 = (a_1 + a_2) \cdot b_4 \\ P_3 = (a_3 + a_4) \cdot b_1 \\ P_4 = a_4 \cdot (b_3 - b_1) \\ P_5 = (a_1 + a_4) \cdot (b_1 + b_4) \\ P_6 = (a_2 - a_4) \cdot (b_3 + b_4) \\ P_7 = (a_1 - a_3) \cdot (b_1 + b_2) \end{array}\right.$$

**Beispiel: Das Auswahlproblem (SELECT)**

Das *Auswahlproblem* lautet wie folgt: *Finde aus dem Array $A[1, \ldots, n]$ das $i$-kleinste Element, mit $1 \leq i \leq n$. O.B.d.A. seien dabei die Elemente $A[1], \ldots, A[n]$ paarweise verschieden.* Dieses Problem ist mit Hilfe eines Sortierverfahrens in $O(n \log n)$ lösbar. Ein einfacher Sortieralgorithmus verfährt wie in Algorithmus 6.

Mit MERGE-SORT als Sortieralgorithmus ergibt sich beispielsweise eine Laufzeit in $\Theta(n \log n)$. Ein Verfahren mit linearer Worst-case-Laufzeit ist SELECT$(n, i)$:

**Laufzeitanalyse von SELECT:**

- Die Schritte **1**, **2**, **4** und **5** sind in $O(n)$.

**Algorithmus 6** : SLOWSELECT$(A, i)$

**Eingabe** : Unsortierte Folge von Zahlen
**Ausgabe** : $i$-kleinstes Element
1 Sortiere die Elemente in aufsteigender Reihenfolge
2 Gib das $i$. Element der sortierten Folge aus

**Algorithmus 7** : SELECT$(A, i)$

**Eingabe** : Unsortiertes Array $A$ von $n$ verschiedenen Zahlen $A = (A[1], \ldots, A[n])$
**Ausgabe** : Das $i$-kleinste Element aus $A$
1 Teile $A$ in $\lfloor \frac{n}{5} \rfloor$ Gruppen mit jeweils 5 Elementen und eine Gruppe mit $n - 5\lfloor \frac{n}{5} \rfloor$ Elementen.
2 Bestimme für jede Gruppe das mittlere Element, sammle diese in $M$.
3 Rufe SELECT$(M, \lceil \frac{n}{10} \rceil)$ auf, mit dem Array $M$ bestehend aus den $\lceil \frac{n}{5} \rceil$ mittleren Elementen, um rekursiv das 'mittlere der mittleren Elemente'$m$ zu bestimmen.
4 $A_1 \leftarrow$ Elemente $a_i$ aus $A$ mit $a_i \leq m$
5 $A_2 \leftarrow$ Elemente $a_j$ aus $A$ mit $a_j > m$
6 Falls $i \leq |A_1|$ rufe SELECT$(A_1, i)$ auf, sonst SELECT$(A_2, i - |A_1|)$

- Schritt **3** benötigt $T(\lceil n/5 \rceil)$ Zeit.
- Schritt **6** benötigt $T(\max(|A_1|, |A_2|)$ Zeit; dabei ist $T(n)$ die Laufzeit von SELECT$(n, i)$ mit $1 \leq i \leq n$.

Dieses Verfahren hat die Laufzeit $T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\max(|A_1|, |A_2|)) + cn$. Man beachte zunächst für $A_2$, dass mindestens die Hälfte aller $\lfloor n/5 \rfloor$ vollen Gruppen mindestens 3 Elemente (die Mehrzahl) zu $A_2$ beisteuern, abzüglich dem einen, mittleren Element. Analoges gilt für $A_1$. Somit lässt sich $\max(|A_1|, |A_2|)$ abschätzen:

$$\begin{aligned} |A_2| &\geq \frac{1}{2} \left\lfloor \frac{n}{5} \right\rfloor 3 - 1 \\ &> \frac{3}{2} \left( \frac{n}{5} - 1 \right) - 1 \\ &= \frac{3}{10} n - \frac{5}{2} \\ \Rightarrow |A_1| &< \frac{7}{10} n + \frac{5}{2} \ . \end{aligned}$$

Analog gilt $|A_1| \geq \frac{1}{2} \lfloor \frac{n}{5} \rfloor 3 > \frac{3}{10} n - 1,5 \Rightarrow |A_2| < \frac{7}{10} n + 1,5$. Also gilt $\max(|A_1|, |A_2|) \leq \frac{7}{10} n + 2,5$ Zunächst stellen wir fest, dass ab $n \geq 9$ gilt: $n > \frac{7}{10} n + 2,5$. Somit muss ab $n < 9$ die Rekursion abgebrochen werden. Es gilt also

$$T(n) \leq \begin{cases} c_0 n & \forall n \leq n_0, n_0 > 9 \text{ geeignet gewählt} \\ T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\frac{7}{10} n + 2,5\right) + c_1 n & \text{für } n > n_0 \end{cases}$$

Wir beweisen $T(n) \in O(n)$ durch Substitution. Sei also $T(n) \leq cn$ für eine Konstante $c > 0$ und alle $n \leq n_0$. Für $n > n_0$ folgt dann per Induktion:

$$\begin{aligned} T(n) &\leq c \left\lceil \frac{n}{5} \right\rceil + c \left( \frac{7}{10} n + 2,5 \right) + c_1 n \\ &\leq c \frac{n}{5} + c + c \frac{7}{10} n + 2,5c + c_1 n \\ &= c \frac{9}{10} n + 3,5c + c_1 n. \end{aligned}$$

Damit garantiert ist, dass $n \cdot c \cdot (9/10) + 3,5 \cdot c + c_1 \cdot n \leq c \cdot n$ gilt, muss $c$ so gewählt werden können, dass gilt $c_1 \cdot n \leq c \cdot \underbrace{(n/10 - 3,5)}_{>0 \text{ nötig}}$. Dazu muss $n > 35$ gelten. In der Abschätzung sollten wir also $n_0 > 35$ wählen.

# Kapitel 1

# Grundlegende Datenstrukturen für Operationen auf Mengen

## 1.1 Union-Find

Das UNION-FIND-Problem ist ein grundlegendes Problem, das sehr viele Anwendungen in sehr unterschiedlichen Bereichen hat. Neben seiner grundsätzlichen Bedeutung zeichnet es sich durch die Tatsache aus, dass es zwar einen ausgesprochen einfachen (einfach zu implementierenden) Algorithmus zu seiner Lösung gibt, die Analyse dieses Algorithmus jedoch „beeindruckend" schwierig ist und überraschenderweise eine fast lineare Laufzeit ergibt (aber eben nur fast) .

**Definition 1.1.** *Das* UNION-FIND-Problem *besteht darin, eine Folge disjunkter Mengen zu verwalten, die sich infolge von Vereinigungsoperationen „laufend" ändert. Gegeben sei eine endliche Grundmenge $M$. Es soll möglichst effizient eine beliebige Abfolge von Operationen folgenden Typs ausgeführt werden:*

- MAKESET$(x)$*: Führe eine neue einelementige Menge $\{x\}$ ein, die zuvor noch nicht existiert hat; $x \in M$.*
- UNION$(S_i, S_j; S_k)$*: Bilde eine neue Menge $S_k := S_i \dot\cup S_j$ aus bisher vorhandenen (disjunkten) Mengen $S_i$ und $S_j$ durch Vereinigung. Entferne $S_i$ und $S_j$.*
- FIND$(x)$*: Für $x \in M$ gib diejenige Menge der bisher entstandenen Mengen an, welche $x$ enthält.*

### 1.1.1 Drei Ansätze

Als Beispiel dienen im Folgenden die Mengen $S_1, S_2,$ und $M$:

$$S_1 = \{1, 3, 5, 7\},\ S_2 = \{2, 4, 8\},\ M = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

#### 1. Ansatz

Benutze ein Array $A$, das die Zuordnung von $x$ zum Index der Menge, die $x$ enthält, angibt.

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| $A[x]$ | 1 | 2 | 1 | 2 | 1 | – | 1 | 2 | – | ... |

**Algorithmus 8** : MAKESET($x$) (#1)

**Eingabe** : Element $x$
**Seiteneffekte** : Neuer Index in $A[x]$
1 Schreibe in $A[x]$ „neuen Index“, etwa $x$

Beispiel: MAKESET(6), $A[6] := 6$. Laufzeit ist in $\mathcal{O}(1)$.

**Algorithmus 9** : FIND($x$) (#1)

**Eingabe** : Element $x$
**Ausgabe** : Menge, in der $x$ enthalten ist
1 Gib $A[x]$ aus

Laufzeit ist in $\mathcal{O}(1)$.

**Algorithmus 10** : UNION($S_i, S_j, S_k$) (#1)

**Eingabe** : Mengen $S_i, S_j, S_k$
**Seiteneffekte** : Mengen $S_i, S_j$ werden zur neuen Menge $S_k$ vereint
1 **Für** $x \in M$
2  **Wenn** $A[x] = i \vee A[x] = j$
3  $A[x] \leftarrow k$

Laufzeit ist in $\mathcal{O}(|M|)$.

Wenn also eine beliebige Folge von $n$ Operationen vom Typ MAKESET, FIND und UNION ausgeführt wird, so ist die Laufzeit dafür in $\mathcal{O}(n^2)$, falls $|M| \in O(n)$. Wir setzen im Folgenden voraus: $|M| \in O(n)$.

### 2. Ansatz

Repräsentiere Mengen durch Wurzelbäume, d.h. jede Menge ist eine „Struktur“ Baum, dessen Knoten die Elemente der Menge sind. Diese Bäume, wie in Abbildung 1.1 entstehen durch die Operationen MAKESET und UNION. Als Mengenindex diene jeweils die Wurzel des Baumes. Dann

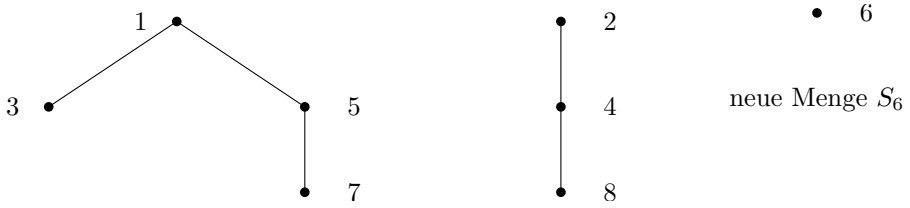



Abbildung 1.1: Einige Mengen, erzeugt durch MAKESET und UNION

werden zwei Mengen vereinigt, indem der eine Baum an die Wurzel des anderen gehängt wird. Als Mengenindex der neuen Menge diene der Index der Menge, an deren Wurzel angehängt wurde, wie in Abbildung 1.2.

$$\begin{array}{lcl} \text{UNION}(S_1, S_2, S_2) & \approx & \text{UNION}(1, 2, 2) \text{ und} \\ \text{UNION}(S_2, S_6, S_6) & \approx & \text{UNION}(2, 6, 6) \end{array}$$

Wir schreiben zur Vereinfachung auch UNION($S_i, S_j$) statt UNION($S_i, S_j, S_k$).

FIND wird für ein Element $x$ ausgeführt, indem von dem entsprechenden Knoten aus durch den Baum bis zu dessen Wurzel gegangen wird. Die *Repräsentation der Bäume* erfolgt durch ein Array

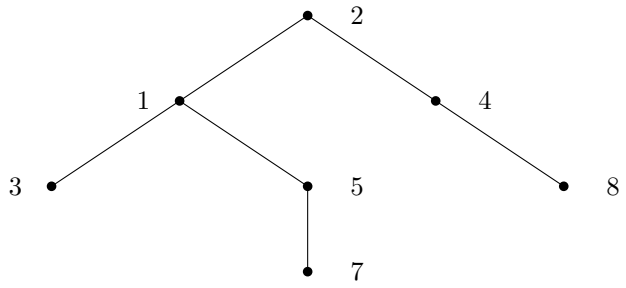



Abbildung 1.2: Resultat von UNION($S_1, S_2, S_2$)

VOR, in dem in VOR[$x$] der *Vorgänger* von $x$ im Baum abgelegt ist. Dabei setzen wir VOR[$x$] := 0, wenn $x$ eine Wurzel ist.

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| VOR[$x$] | 2 | 0 | 1 | 2 | 1 | 0 | 5 | 4 | – | ... |

**Algorithmus 11** : FIND($x$) (#2)

**Eingabe** : Element $x$
**Ausgabe** : Menge, in der $x$ enthalten ist

```
1 j ← x
2 Solange VOR[j] ≠ 0 (und definiert) tue
3   j ← VOR[j]
4 Gib j aus
```

Die Laufzeit ist in $\mathcal{O}(n)$.

**Algorithmus 12** : UNION($i, j$) (#2)

**Eingabe** : Mengen $S_i, S_j$
**Seiteneffekte** : Elemente der Menge $S_i$ werden zur Menge $S_j$ hinzugefügt

```
1 VOR[i] ← j
```

**Algorithmus 13** : MAKESET($x$) (#2)

**Eingabe** : Element $x$
**Seiteneffekte** : Neuer Index in $A[x]$

```
1 VOR[x] ← 0
```

Die Laufzeit für UNION und MAKESET ist in $\mathcal{O}(1)$.

**Gesamtlaufzeit:** Bei einer Folge von $n$ Operationen vom Typ MAKESET, UNION und FIND ist die Gesamtlaufzeit in $\Theta(n^2)$:

- $\Theta(n)$ Operationen vom Typ MAKESET (z.B $n/4$), gefolgt von
- $\Theta(n)$ Operationen vom Typ UNION, gefolgt von
- $\Theta(n)$ Operationen vom Typ FIND.

Operationen vom Typ FIND sind also besonders teuer, da durch UNION „sehr hohe“ Bäume entstehen können, wie in Abbildung 1.3 zu sehen.

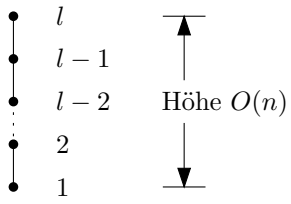



Abbildung 1.3: Tiefe eines Baumes

Eine Folge von $t$ verschiedenen FIND hat dann eine Laufzeit von mindestens

$$\sum_{i=1}^{t} i \in \Theta(t^2) = \Theta(n^2)$$

(weil $t \in \Theta(n)$).

### 3. Ansatz

Wie nehmen zwei Modifikationen am zweiten Ansatz vor:

- Modifikation 1: ein Ausgleich bei UNION: „weighted UNION rule“ oder „balancing“.
- Modifikation 2: „Pfadkompression“ bei FIND.

**Modifikation 1: „weighted UNION“:** Bei der Ausführung von UNION wird immer der „kleinere“ Baum an den „größeren“ Baum gehängt, wie in Abbildung 1.4. „Kleiner“ bzw. „größer“ bezieht sich hier einfach auf die Anzahl der Knoten (Kardinalität der entsprechenden Menge). Um zu entscheiden, welcher Baum kleiner bzw. größer ist, wird in VOR$[x]$ für Wurzeln $x$ jeweils die Knotenzahl des Baums als negative Zahl gespeichert. Das negative Vorzeichen kennzeichnet die Wurzeleigenschaft und der Betrag die Kardinalität der Menge.

**Formal:** UNION$(i, j)$**:** Es gilt:

- VOR$[i] = -\#$(Knoten im Baum i)
- VOR$[j] = -\#$(Knoten im Baum j)

**Algorithmus 14** : UNION$(i, j)$ (#3)

```
1 z ← VOR[i] + VOR[j]
2 Wenn |VOR[i]| < |VOR[j]|
3   VOR[i] ← j und VOR[j] ← z
4 sonst
5   VOR[j] ← i und VOR[i] ← z .
```

Die Laufzeit ist offensichtlich in $\Theta(1)$. Lässt sich nun etwas über die Höhe von Bäumen aussagen, die durch „weighted UNION“ entstanden sind?

**Lemma 1.2.** *Entsteht durch eine Folge von* MAKESET *und weighted* UNION *über einer Menge $M$ ein Baum $T$ mit $|T| = t$ ($T$ enthält $t$ Knoten), so ist $h(T) \leq \log_2 t$, wobei $h(T)$ die Höhe von $T$ ist, also die maximale Anzahl von Kanten auf einem einfachen Weg von der Wurzel von $T$ zu einem Blatt von $T$.*

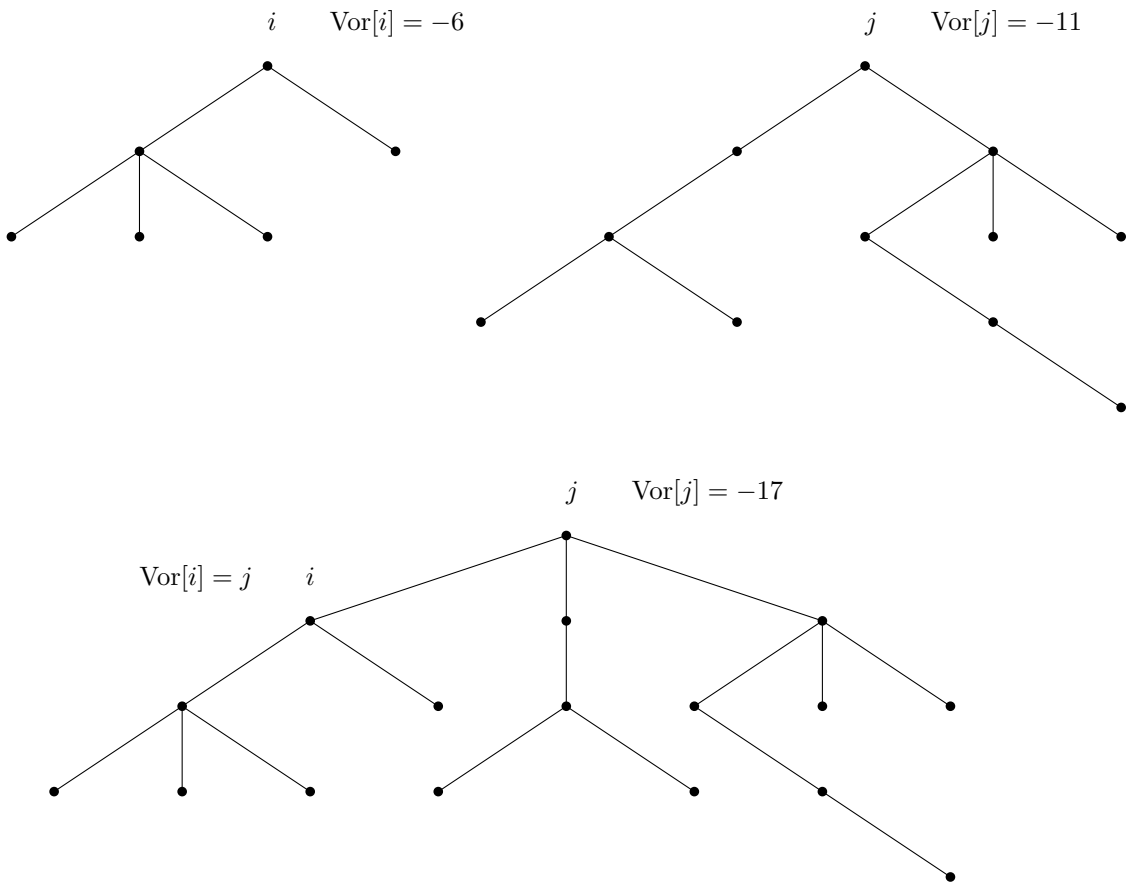



Abbildung 1.4: „weighted UNION“: UNION$(i, j)$

*Beweis.* Induktion über die Anzahl der UNION-Operationen.

Solange keine UNION-Operation ausgeführt wurde, haben alle Bäume Höhe 0 und einen Knoten, d.h. es gilt für alle $T$:

$$h(T) = 0 \leq \log_2 1 = 0.$$

Betrachte die $n$-te UNION-Operation, etwa UNION$(i, j)$, wobei $T_i$ Baum mit Wurzel $i$ und $T_j$ Baum mit Wurzel $j$ vor Ausführung der Operation UNION$(i, j)$ sei. O.B.d.A. sei $|T_j| \geq |T_i|$. Dann wird $T_i$ durch UNION$(i, j)$ an die Wurzel $j$ von $T_j$ angehängt und es entsteht der Baum $T_{\text{UNION}(i,j)}$ mit

$$h(T_{\text{UNION}(i,j)}) = \max(h(T_j), h(T_i) + 1).$$

- Falls $h(T_j) > h(T_i) + 1$, dann ist

$$h(T_{\text{UNION}(i,j)}) = h(T_j) \leq \log_2(|T_j|) \leq \log_2(|T_{\text{UNION}(i,j)}|).$$

- Falls $h(T_j) \leq h(T_i) + 1$, dann ist

$$h(T_{\text{UNION}(i,j)}) = h(T_i) + 1 \leq \log_2(|T_i|) + 1 \leq \log_2(2|T_i|) \leq \log_2(|T_{\text{UNION}(i,j)}|) \ . \qquad \square$$

**Beobachtung 1.3.** *In einem durch weighted* UNION*-Operationen entstandenen Baum kann jede* FIND*-Operation in $\mathcal{O}(\log n)$ Zeit ausgeführt werden. Für eine Folge von $n$* MAKESET*-,* UNION*- und* FIND*-Operationen ergibt sich dann eine Gesamtlaufzeit von $\mathcal{O}(n \log n)$.*

**Modifikation 2: Pfadkompression:** Bei der *Pfadkompression* werden bei der Ausführung einer Operation FIND($i$) alle Knoten, die während dieses FIND durchlaufen werden, zu direkten Nachfolgern der Wurzel des Baumes gemacht, in dem Knoten $i$ liegt. Abbildung 1.5 illustriert wie Pfadkompression entsprechend Algorithmus 15 erfolgt.

**Algorithmus 15** : FIND($x$) (#3)

**Eingabe** : Element $x$
**Ausgabe** : Menge, in der $x$ enthalten ist

```
1 j ← x
2 Solange VOR[j] > 0 tue
3 |  j ← VOR[j]
4 i ← x
5 Solange VOR[i] > 0 tue
6 |  temp ← i
7 |  i ← VOR[i]
8 |  VOR[temp] ← j
9 Gib j aus
```

Auch bei FIND mit Pfadkompression hat eine Operation FIND eine Worst-case-Laufzeit von $\mathcal{O}(\log n)$. Die Gesamtlaufzeit ist also weiterhin in $\mathcal{O}(n \log n)$. Eine genauere amortisierte Analyse ergibt jedoch eine bessere Gesamtlaufzeit von $\mathcal{O}(n \cdot G(n))$, wobei $G(n)$ sehr, sehr langsam wächst – wesentlich langsamer als $\log n$. $G(n)$ ist für alle praktisch relevanten Werte $n$ „klein“, d.h. $G(n)$ verhält sich „praktisch“ wie eine Konstante.

**Definition 1.4.**

$$G(n) := min\{y : F(y) \geq n\},$$

*wobei* $F(0) := 1$ *und* $F(y) = 2^{F(y-1)}$ *für* $y > 0$.

Wie schnell wächst $F(n)$ bzw. $G(n)$?

| $F(0)$ | $F(1)$ | $F(2)$ | $F(3)$ | $F(4)$ | $F(5)$ |
|---|---|---|---|---|---|
| | | $= 2^2$ | $= 2^4$ | $= 2^{16}$ | $= 2^{65536}$ |
| $= 1$ | $= 2$ | $= 4$ | $= 16$ | $= 65536$ | |

Für alle praktisch relevanten Werte von $n$ ist also $G(n) \leq 5$. Man schreibt oft auch $\log^*(n)$ für $G(n)$.

### 1.1.2 Die Laufzeit von Union-Find

**Satz 1.5 (Hopcroft & Ullman 1973).** *Die Gesamtlaufzeit für eine Folge $Q$ von $n$ Operationen vom Typ* MAKESET*, weighted* UNION *und* FIND *mit Pfadkompression ist in $\mathcal{O}(n \cdot G(n))$.*

Für eine beliebige Folge $Q$ von Operationen trennen wir die Laufzeit für die MAKESET- und UNION-Operationen von der Laufzeit für die FIND-Operationen. Die Gesamtlaufzeit für alle MAKESET- und alle UNION-Operationen ist in $\mathcal{O}(n)$.

Die Gesamtlaufzeit für alle FIND-Operationen ist im wesentlichen proportional zu der Anzahl der durch FIND-Operationen bewirkten „Knotenbewegungen“ (Zuteilung eines neuen Vorgängers). Eine einzelne solche Knotenbewegung hat Aufwand $\mathcal{O}(1)$.

Die Knotenbewegungen werden in zwei Klassen $A$ und $B$ aufgeteilt. Dazu werden *Ranggruppen* $\gamma_1, \ldots, \gamma_{G(n)+1}$ gebildet, für die sich die jeweilige Anzahl an Knotenbewegungen „leicht“ aufsummieren lässt. Im Folgenden dieses Abschnitts beweisen wir Satz 1.5.

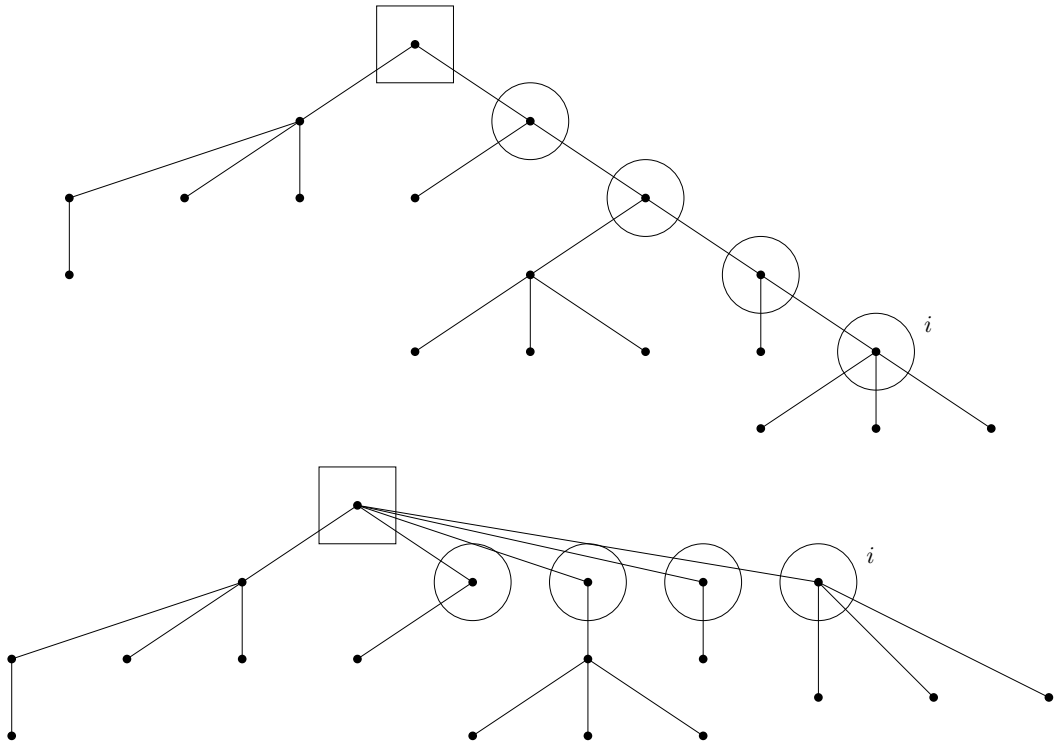



Abbildung 1.5: Pfadkompression bei FIND

**Der Rang $r(v)$**

**Definition 1.6 (Rang).** *Der* Rang $r(v)$ *eines Knotens $v$ sei definiert als die Höhe des Unterbaums mit Wurzel $v$ im Baum $T$, der nach Ausführung aller* MAKESET- *und* UNION-*Operationen aus $Q$* ohne *die* FIND-*Operationen entstehen würde (und der $v$ enthält).*

Nenne diese Folge von Operationen $Q'$. Wenn also $T(v)$ der Unterbaum von $T$ mit Wurzel $v$ ist, so ist $r(v) = h(T(v))$. Beachte, dass der Rang $r(v)$ des Knotens $v$ *nicht* zu jeder Zeit während $Q$ dem Wert $h(T(v))$ entspricht, da $v$ durch Pfadkompression seinen gesamten Unterbaum verlieren kann. Für die Bestimmung von $r(v)$ jedoch betrachten wir *keine* Pfadkompression. Abbildung 1.6 zeigt wie sich der Rang eines Knoten darstellt. Im Folgenden bezeichnen wir, wie bei Bäumen üblich, den Vorgänger eines Knotens $v$ mit $p[v]$.

**Beobachtung.**

1. *Für alle Knoten $v$ gilt:* $r(v) < r(p[v])$.
2. *Es ist $r(p[v])$ (der Rang der Vorgängerknoten von $v$) monoton steigend.*
3. *Für alle Wurzeln $x$ gilt:* $|T(x)| \geq 2^{r(x)}$.
4. $|\{$*Knoten mit Rang $r$*$\}| \leq \frac{n}{2^r}$.
5. *Für alle Knoten $v$ gilt:* $r(v) \leq \lfloor \log n \rfloor$.

*Beweis.*

- 1. und 2. sind offensichtlich.
- 3.: Beachtet man, dass gilt $r(v) \geq h(v)$ so ist dies mit Hilfe von Lemma 1.2 offenbar.

- 4.: Für alle Knoten $v$ gilt $r(p[v]) > r(v)$. Also gilt für $v_1 \neq v_2$ mit $r(v_1) = r(v_2) = r$, dass $v_1 \notin T(v_2)$ und $v_2 \notin T(v_1)$ und somit $T(v_1) \cap T(v_2) = \emptyset$. Also sind die Unterbäume aller Knoten mit Rang $r$ disjunkt und es folgt |Knoten mit Rang $r$| $\leq \frac{n}{2^r}$.

  Alternativ lässt sich hier auch mit Lemma 1.2 argumentieren.

- 5. ergibt sich als Korollar zu 3. Es folgt somit, dass stets $r(v) \leq \log n$ ist.

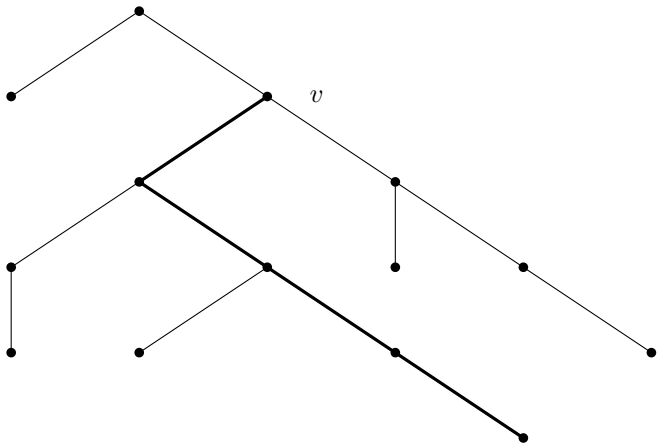



Abbildung 1.6: Der Rang von Knoten $v$ ist $r(v) = 4$

**Die Ranggruppe $\gamma_i$**

**Definition 1.8 (Ranggruppe).** *Für $j \geq 1$ definiere*

$$\gamma_j := \{v : \log^{(j+1)} n < r(v) \leq \log^{(j)} n\},$$

*wobei gilt:*

$$\log^{(j)} n := \begin{cases} n & \text{falls } j = 0, \\ \log(\log^{(j-1)} n) & \text{falls } j > 0 \text{ und } \log^{(j-1)} n > 0, \\ \text{undefiniert} & \text{falls } j > 0 \text{ und } \log^{(j-1)} n \leq 0 \text{ oder} \\ & \log^{(j-1)} n \text{ undefiniert} \end{cases}$$

*und*

$$\gamma_j := \begin{cases} \emptyset & \text{wenn } \log^{(j)} n \text{ undefiniert}, \\ \{v : r(v) = 0\} & \text{falls } 0 \leq \log^{(j)} n < 1. \end{cases}$$

*$\gamma_j$ heißt die $j$-te* Ranggruppe *und es gilt*

$$\begin{aligned} \gamma_1 &= \{v : \log^{(2)} n < r(v) \leq \log n\} \\ &\vdots \\ \gamma_{G(n)+1} &= \emptyset, \quad \text{da für } v \in \gamma_{G(n)+1} \text{ gilt: } r(v) < -x, \text{ da } \log^{(G(n))} n \leq 1, \\ \gamma_{G(n)+2} &= \emptyset, \quad \text{da } \log^{(G(n)+2)} n \text{ undefiniert ist.} \end{aligned}$$

**Beispiel 1.9.** Ranggruppen für $n = 66000$

$$\begin{aligned}
\gamma_1 &= \{v : \log^{(2)} 66000 < r(v) \leq \log 66000\} = \{v : 4 < r(v) \leq 16\} \\
\gamma_2 &= \{v : \log^{(3)} 66000 < r(v) \leq \log^{(2)} 66000\} = \{v : 2 < r(v) \leq 4\} \\
\gamma_3 &= \{v : \log^{(4)} 66000 < r(v) \leq \log^{(3)} 66000\} = \{v : 1 < r(v) \leq 2\} \\
\gamma_4 &= \{v : \log^{(5)} 66000 < r(v) \leq \log^{(4)} 66000\} = \{v : 0 < r(v) \leq 1\} \\
\gamma_5 &= \{v : \log^{(6)} 66000 < r(v) \leq \log^{(5)} 66000\} = \{v : -x < r(v) \leq 0\} \\
&= \{v : r(v) = 0\}
\end{aligned}$$

Es ist $G(66000) = 5$ da $\underbrace{2^{\left(2^{(2^2)}\right)}}_{4} = 65536 \leq 66000 \leq 2^{65536} = \underbrace{2^{\left(2^{\left(2^{(2^2)}\right)}\right)}}_{5}$ und somit ist $\gamma_{G(66000)+1} = \emptyset$. Die Ranggruppe $\gamma_{G(n)+1}$ ist nur dann nicht leer, falls $n$ eine Zweierpotenz ist. ∎

Als Hilfsmittel zeigen wir zunächst eine Beobachtung zu den Ranggruppen:

**Beobachtung 1.10 (Größe der Ranggruppen).** *Für die Menge der Knoten in Ranggruppe $\gamma_j$ gilt:* $|\gamma_j| \leq \frac{2n}{\log^{(j)} n}$

*Beweis.* Wir benutzen die Abschätzung 4 aus Beobachtung 1.7:

$$\begin{aligned}
|\gamma_j| &= \sum_{\text{Ränge i in } \gamma_j} |\text{Knoten mit Rang } i| \leq \sum_{i=\lceil \log^{(j+1)} n \rceil}^{\lfloor \log^{(j)} n \rfloor} \frac{n}{2^i} \\
&= \sum_{i=\lceil \log^{(j+1)} n - \log^{(j+1)} n \rceil}^{\lfloor \log^{(j)} n - \log^{(j+1)} n \rfloor} \frac{n}{2^{i+\log^{(j+1)} n}} = \sum_{i=0}^{\lfloor \log^{(j)} n - \log^{(j+1)} n \rfloor} \frac{n}{2^i} \cdot \frac{1}{2^{\log^{(j+1)} n}} \\
&\leq \frac{n}{2^{\log^{(j+1)} n}} \Big( \underbrace{\sum_{i=0}^{\infty} \frac{1}{2^i}}_{=2} \Big) \\
&= \frac{2n}{2^{\log^{(j+1)} n}} = \frac{2n}{2^{\log(\log^{(j)} n)}} = \frac{2n}{\log^{(j)} n}
\end{aligned}$$

□

### Die Analyse

Die Klasse aller Knotenbewegungen, die durch FIND-Operationen ausgeführt werden, wird aufgeteilt in zwei disjunkte Klassen $A$ und $B$.

- Klasse $A$: diejenigen Knotenbewegungen, die für Knoten $v$ ausgeführt werden, deren Vorgänger $p[v]$ einer anderen Ranggruppe angehört (Vorgänger zum Zeitpunkt der entsprechenden FIND-Operation).
- Klasse $B$: diejenigen Knotenbewegungen, die für Knoten $v$ ausgeführt werden, deren Vorgänger $p[v]$ zu derselben Ranggruppe gehört.

Bei der nun folgenden *amortisierten Analyse* werden bei Knotenbewegungen aus $A$ die Kosten der entsprechenden FIND-Operation zugeordnet. Zur Abschätzung dient dann dieser Wert multipliziert mit der Anzahl FIND-Operation. Bei Knotenbewegungen aus $B$ werden Kosten „dem bewegten Knoten“ zugeordnet. Hier wird dann argumentiert, dass jeder Knoten nur eine begrenzte Anzahl Klasse $B$ Knotenbewegungen erfährt und die Summe dieser, über alle Knoten, in $\mathcal{O}(n \cdot G(n))$ ist.

**Klasse $A$ Knotenbewegungen:** Betrachte die Ausführung von FIND($v$). Auf dem Pfad von $v$ zur Wurzel des entsprechenden Baumes steigt der Rang nach Beobachtung 1.7 monoton an und somit auch die Ranggruppe. Da es maximal $G(n)+1$ Ranggruppen gibt, gibt es auf diesem Pfad höchstens $G(n)$ Knoten, deren direkter Vorgänger in einer anderen Ranggruppe liegt. Die Operation FIND($v$) verursacht also höchstens $G(n)$ Knotenbewegungen aus Klasse $A$. Die Gesamtkosten der Klasse-$A$ Knotenbewegungen liegen also in $O(n \cdot G(n))$.

**Klasse $B$ Knotenbewegungen:** Die Anzahl verschiedener Ränge in Ranggruppe $\gamma_j$ kann nach oben abgeschätzt werden durch $\log^{(j)} n \geq \log^{(j)} n - \log^{(j+1)} n$ (siehe Definition 1.8). Ein Knoten aus $\gamma_j$ kann somit höchstens $\log^{(j)} n$ mal bewegt werden, bevor er einen Vorgänger erhält, der in einer anderen Ranggruppe $\gamma_i$ mit $i < j$ liegt. Wegen der Monotonizität der Ranggruppen nach Beobachtung 1.7 wird sich dies auch nicht mehr zu einem späteren Zeitpunkt ändern. Damit ist die Gesamtzahl an Knotenbewegungen der Klasse $B$ für Knoten der Ranggruppe $\gamma_j$ höchstens

$$
\begin{aligned}
|\text{Klasse-}B \text{ Bewegungen in } \gamma_j| &\leq |\gamma_j| \cdot log^{(j)} n && (1.1)\\
&\leq \frac{2n}{\log^{(j)} n} \cdot \log^{(j)} n && (1.2)\\
&= 2n. && (1.3)
\end{aligned}
$$

Da insgesamt höchstens $G(n)+1$ nicht leere Ranggruppen existieren, ist die Gesamtlaufzeit für Knotenbewegungen der Klasse $B$ also $(G(n)+1) \cdot 2n \in \mathcal{O}(n \cdot G(n))$.

**Gesamtlaufzeit:** Insgesamt ist somit die Gesamtlaufzeit für alle durch FIND ausgelösten Knotenbewegungen ebenfalls in $\mathcal{O}(n \cdot G(n))$. ■

### 1.1.3 Bemerkungen

Eine genauere Analyse führt zu einer Laufzeit aus $\mathcal{O}(m \cdot \alpha(m,n))$ für $m$ FIND, UNION und MAKESET-Operationen mit $n$ Elementen (Tarjan, 1975). Dabei ist

$$\alpha(m,n) := \min\{i \geq 1 : A(i, \lfloor \tfrac{m}{n} \rfloor) > \log n\}$$

und

$$
\begin{aligned}
A(1,j) &= 2^j \text{ für } j \geq 1\\
A(i,1) &= A(i-1,2) \text{ für } i \geq 2\\
A(i,j) &= A(i-1, A(i,j-1)) \text{ für } i,j \geq 2.
\end{aligned}
$$

Die Funktion $A$ heißt *Ackermann-Funktion* und wächst noch stärker als $F$ (iterative Zweierpotenz). Andererseits wurde bewiesen, dass die Laufzeit für eine Folge von $m$ FIND und $n$ UNION- und MAKESET-Operationen im Allgemeinen auch in $\Omega(m \cdot \alpha(m,n))$ liegt (Tarjan, 1979).

Für spezielle Folgen von UNION-, FIND- und MAKESET-Operationen, über die man vorab „gewisse strukturelle Informationen“ hat, ist die Laufzeit in $\mathcal{O}(m)$ (Gabow & Tarjan, 1985).

Es lassen sich oft Algorithmen oder Teile von Algorithmen als Folgen von UNION-, FIND- und MAKESET-Operationen auffassen und sind damit „fast“ in linearer Zeit durchführbar. Oft tritt sogar der Fall auf, dass die Folge „spezielle Struktur“ hat und wirklich in Linearzeit ausführbar ist.

## 1.2 Anwendungsbeispiele für Union-Find

### 1.2.1 Der Algorithmus von Kruskal für MST

Die Abkürzung *MST* steht für *Minimum Spanning Tree*, siehe dazu auch Kapitel 2. Algorithmus 16 erstellt einen MST eines Graphen in Form einer Kantenmenge. Im nächsten Kapitel beschreibt Algorithmus 27 diesen Algorithmus informell. Wenn SORT($E$) bereits vorliegt, dann ist die Laufzeit

**Algorithmus 16** : Algorithmus von Kruskal

**Eingabe** : Knotenliste $V$, Kantenliste $E$ mit Kantengewichten.
**Ausgabe** : Datenstruktur: Menge GRÜN.

```
1 GRÜN ← ∅
2 Sortiere E entsprechend Gewicht „aufsteigend“, die sortierte Liste sei SORT(E)
3 Für v ∈ V
4 |  MAKESET(v)
5 Für {v, w} ∈ SORT(E)
6 |  Wenn FIND(v) ≠ FIND(w)
7 |  |  UNION(FIND(v), FIND(w))
8 |  |  GRÜN ← GRÜN ∪ {{v, w}}
9 Gib GRÜN aus
```

in $\mathcal{O}(|E| \cdot \alpha(|E|, |V|))$.

### 1.2.2 Das OFFLINE-MIN-Problem

**Problem.** *Gegeben sei eine Menge $M = \emptyset$. Führe eine Folge $Q$ von $n$ Operationen vom Typ*

$$\begin{array}{ll} \text{INSERT}[i]: & M := M \cup \{i\} \quad \textit{oder} \\ \text{EXTRACT-MIN}: & M := M \setminus \{\min M\} \end{array}$$

*aus, wobei $i$ aus der Menge $\{1, \ldots, n\}$ ist. Eine Operation* INSERT$[i]$ *trete für jedes $i$ höchstens einmal in der Folge auf.*

*Zu einer Folge $Q$ wollen wir alle $i$ finden, die durch eine Operation* EXTRACT-MIN *entfernt werden und die entsprechende* EXTRACT-MIN*-Operation angeben (daher Bezeichnung „Offline“: $Q$ ist vorher bekannt).*

Wir lösen das Problem durch eine Folge von UNION- und FIND-Operationen.

**Bemerkung 1.11.** *Eine Folge von $n$ Operationen vom Typ* UNION *und* FIND*, beginnend mit einer Menge disjunkter Teilmengen einer Menge mit $\mathcal{O}(n)$ Elementen, ist natürlich ebenfalls in $\mathcal{O}(n \cdot G(n))$ (bzw. $\mathcal{O}(n \cdot \alpha(n, n))$) ausführbar.*

Schreibe die Folge $Q$ als

$$Q_1 E Q_2 E \ldots E Q_{k+1},$$

wobei $Q_j$ für $1 \leq j \leq k+1$ nur aus INSERT-Operationen besteht (möglicherweise $Q_j = \emptyset$); $E$ stehe für eine EXTRACT-MIN-Operation. Die Anzahl der EXTRACT-MIN sei also $k$.

Für den UNION-FIND-Algorithmus initialisieren wir (paarweise disjunkte) Mengen

$$M_j := \{i : \text{INSERT}[i] \text{ liegt in } Q_j\}$$

für $1 \leq j \leq k+1$. Benutze Arrays PRED (predecessor) und SUCC (successor) zur Erzeugung doppelt verketteter Listen der Werte $j$, für die eine Menge $M_j$ existiert.

Zu Beginn sei $\textsc{Pred}[j] = j - 1$ für $2 \leq j \leq k+1$ und $\textsc{Succ}[j] = j + 1$ für $1 \leq j \leq k$.

**Formale Beschreibung der** Offline-Min**-Prozedur.**

**Algorithmus 17** : Offline-Min

```
1 Für i = 1 bis n
2 |  j ← Find[i]
3 |  Wenn j ≤ k
4 |  |  Schreibe „i ist entfernt worden im j-ten Extract-Min “
5 |  |  Union(j, Succ[j], Succ[j])
6 |  |  Succ[Pred[j]] ← Succ[j]
7 |  |  Pred[Succ[j]] ← Pred[j]
```

**Beispiel 1.12.** Folge $Q$: $\underbrace{4,3}_{Q_1}, E, \underbrace{2}_{Q_2}, E, \underbrace{1}_{Q_3}, E, \underbrace{\quad}_{Q_4}$ mit $k = 3$.

Mengen: $1 = \{4,3\}, 2 = \{2\}, 3 = \{1\}, 4 = \emptyset$.

i=1: $\textsc{Find}[1] = 3; 3 \leq k = 3 \rightsquigarrow$ „1 ist im 3-ten Extract-Min entfernt worden.“ $\rightsquigarrow 1 = \{4,3\}, 2 = \{2\}, 4 = \{1\}$,
$\textsc{Succ}[2] = 4$, $\textsc{Pred}[4] = 2$

i=2: $\textsc{Find}[2] = 2; 2 \leq 3 \rightsquigarrow$ „2 ist im 2-ten Extract-Min entfernt worden“ $\rightsquigarrow 1 = \{4,3\}, 4 = \{1,2\}$,
$\textsc{Succ}[1] = 4$, $\textsc{Pred}[4] = 1$

i=3: $\textsc{Find}[3] = 1; 1 \leq 3 \rightsquigarrow$ „3 ist im 1-ten Extract-Min entfernt worden.“ $\rightsquigarrow 4 = \{4,3,1,2\}$,
$\textsc{Succ}[0] = 4$, $\textsc{Pred}[4] = 0$

i=4: $\textsc{Find}[4] = 4; 4 > 3 \rightsquigarrow$ „4 ist nicht gelöscht worden.“ ∎

**Bemerkung 1.13.** *Der Algorithmus* Offline-Min *ist sogar in $\mathcal{O}(n)$, da die* Union-Find*-Folge zu den Spezialfällen gehört, die in Linearzeit ausführbar sind.*

### 1.2.3 Äquivalenz endlicher Automaten

**Definition 1.14.** *Ein endlicher Automat $\mathcal{A}$ besteht aus einem Alphabet $\Sigma$, einer endlichen Zustandsmenge $Q$, einem Anfangszustand $q_0 \in Q$, einer Menge von Endzuständen $F \subseteq Q$ und der „Zustandsübergangsfunktion“ $\delta : Q \times \Sigma \longrightarrow Q$. $\Sigma^*$ bezeichne die Menge aller Wörter endlicher Länge über $\Sigma$ (inkl. dem „leeren Wort“ $\varepsilon$). Dann lässt sich $\delta$ erweitern zu $\delta : Q \times \Sigma^* \longrightarrow Q$ durch*

$$\delta(q, \varepsilon) := q \text{ und}$$
$$\delta(q, wa) := \delta(\delta(q, w), a)$$

*für alle $w \in \Sigma^*$ und $a \in \Sigma$ und $q \in Q$.*

*Der Automat $\mathcal{A}$* akzeptiert *ein Wort $w$ genau dann, wenn $\delta(q_0, w) \in F$. Die Sprache der Wörter, die von $\mathcal{A}$ akzeptiert werden, heißt $L(\mathcal{A})$. Zwei Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ heißen* äquivalent*, wenn $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ ist. Schreibe dann $\mathcal{A}_1 \equiv \mathcal{A}_2$.*

**Beispiel 1.15.**

$\Sigma = \{0,1\}, Q = \{q_0, q_1, q_2, q_3\}, F = \{q_0\}$,
$L(\mathcal{A}) := \{w \in \Sigma^* :$ Anzahl der 0 in $w$ und Anzahl der 1 in $w$ sind gerade$\}$. ∎

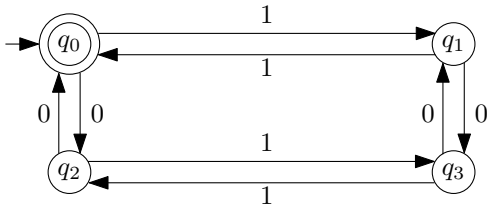



Abbildung 1.7: Deterministischer endlicher Automat $\mathcal{A}$

Wir wollen möglichst „effizient“ für zwei beliebige endliche Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ über dem Alphabet $\Sigma$ entscheiden, ob sie äquivalent sind, d.h., ob $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ ist.

**Beispiel 1.16.** $\Sigma = \{0,1\}$.
$\mathcal{A}_1 : Q_1 = \{a, b, c, d, e, f, g, h\}$, Anfangszustand $a$, Endzustandsmenge $\{c\}$.

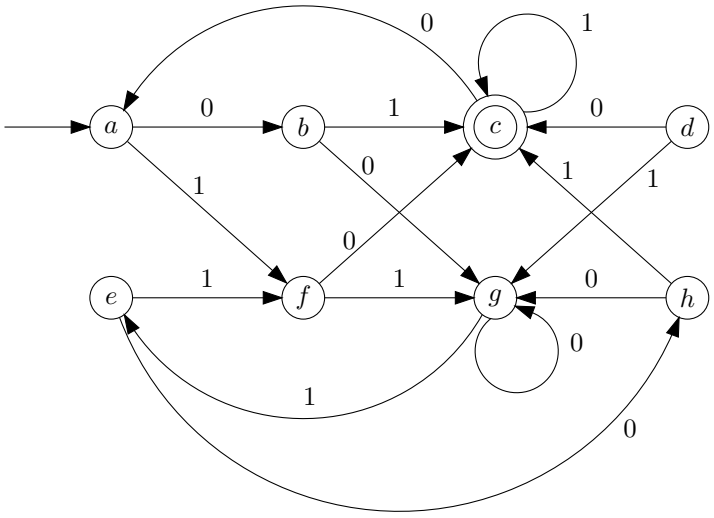



Abbildung 1.8: Deterministischer endlicher Automat $\mathcal{A}_1$

$\mathcal{A}_2 : Q_2 = \{v, w, x, y, z\}$, Anfangszustand $v$, Endzustandsmenge $\{y\}$. Man betrachte die deterministischen endlichen Automaten in Abbildungen 1.8 und 1.9. Sind $\mathcal{A}_1$ und $\mathcal{A}_2$ äquivalent? Die Antwort ist „ja“. Wie kann man das aber testen? ∎

Wir benutzen zur Entscheidung, ob $\mathcal{A}_1 \equiv \mathcal{A}_2$, den Begriff *äquivalenter Zustände*: schreibe $q \equiv q'$. Aus der Theoretischen Informatik ist dieser Begriff für Zustände desselben Automaten $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ bekannt. Dort wurde definiert, dass

$$q \equiv q' \text{ für } q, q' \in Q, \text{ wenn für alle } w \in \Sigma^* \text{ gilt}$$
$$\delta(q, w) \in F \iff \delta(q', w) \in F.$$

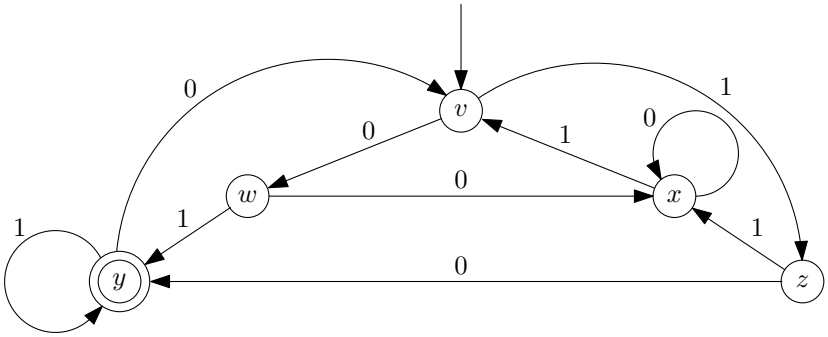



Abbildung 1.9: Deterministischer endlicher Automat $\mathcal{A}_2$

Entsprechend definiere für zwei endliche Automaten $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ und $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$, dass für Zustände $q_1 \in Q_1, q_2 \in Q_2$ mit o.B.d.A. $Q_1 \cap Q_1 = \emptyset$ gilt:

$$q_1 \equiv q_2 \text{ wenn } \delta_1(q_1, w) \in F_1 \iff \delta_2(q_2, w) \in F_2 \text{ für alle } w \in \Sigma^*.$$

Dann gilt offensichtlich: $\mathcal{A}_1 \equiv \mathcal{A}_2 \iff s_1 \equiv s_2$.

Es ist leicht zu sehen (siehe auch Theoretische Informatik), dass gilt:

$$q \equiv q' \text{ für } q, q' \in Q \text{ genau dann, wenn } \delta(q, a) \equiv \delta(q', a) \text{ für alle } a \in \Sigma.$$

Außerdem ist $q \not\equiv q'$ für alle $q \in F$, $q' \in Q \setminus F$ (betrachte das leere Wort $\varepsilon$).

Entsprechend gilt auch:

$$q_1 \equiv q_2 \text{ für } q_1 \in Q_1, q_2 \in Q_2 \text{ genau dann, wenn } \delta(q_1, a) \equiv \delta(q_2, a) \text{ für alle } a \in \Sigma.$$

Außerdem ist $q_1 \not\equiv q_2$ für alle $q_1 \in F_1$ und $q_2 \in Q_2 \setminus F_2$ (bzw. $q_1 \in Q_1 \setminus F_1$ und $q_2 \in F_2$).

Diese Eigenschaften benutzen wir nun, um zu testen, ob $\mathcal{A}_1 \equiv \mathcal{A}_2$, d.h. $s_1 \equiv s_2$. Wir nehmen an, dass $s_1 \equiv s_2$, und folgern daraus die Äquivalenz weiterer Zustände. Auf diese Weise erhalten wir eine Partition von $Q_1 \cup Q_2$ in Klassen „angeblich äquivalenter“ Zustände. Enthält eine dieser Klassen sowohl einen Endzustand als auch einen Nichtendzustand, so kann auch nicht $s_1 \equiv s_2$ gelten.

### Vorgehensweise informell

In einem Stack $S$ werden Paare von Zuständen $(q_1, q_2)$ gehalten, die „angeblich“ äquivalent sind, deren Nachfolger $(\delta_1(q_1, a), \delta_2(q_2, a))$ usw. noch nicht betrachtet wurden. Zu Beginn enthält $S$ das Paar $(s_1, s_2)$. Um eine Partition von $Q_1 \cup Q_2$ in Klassen „angeblich“ äquivalenter Zustände zu berechnen, wird eine Folge von Union- und Find-Operationen benutzt. Beginnend mit den einelementigen Teilmengen von $Q_1 \cup Q_2$ werden jeweils die Mengen vereinigt, die $q_1$ bzw. $q_2$ enthalten für ein Paar $(q_1, q_2)$, das als „angeblich“ äquivalent nachgewiesen wurde. Dann ist $\mathcal{A}_1 \equiv \mathcal{A}_2$ genau dann, wenn die Partition von $Q_1 \cup Q_2$, mit der das Verfahren endet, keine Menge enthält, die sowohl einen Endzustand als auch einen Nichtendzustand als Element enthält.

### Formale Beschreibung

### Laufzeitanalyse

Sei $n := |Q_1| + |Q_2|$ und $|\Sigma| = k$. Da zu Beginn die Partition aus $n$ Mengen besteht, werden höchstens $n - 1$ Union ausgeführt. Die Anzahl der Find ist proportional zur Anzahl der Paare,

**Algorithmus 18** : Äquivalenz endlicher Automaten

```
1  S ← ∅
2  Für q ∈ Q_1 ∪ Q_2
3      MAKESET(q)
4  PUSH((s_1, s_2))
5  Solange S ≠ ∅ tue
6      (q_1, q_2) ← POP(S)
7      Wenn FIND[q_1] ≠ FIND[q_2]
8          UNION(FIND[q_1], FIND[q_2])
9          Für a ∈ Σ
10             PUSH(δ_1(q_1, a), δ_2(q_2, a))
```

die insgesamt auf den Stack gelegt werden. Dies sind höchstens $k \cdot (n-1) + 1$ Paare, da nur nach jeder UNION-Operation jeweils $k$ Paare auf $S$ gelegt werden. Wird $|\Sigma| = k$ als Konstante angenommen, so ist die Laufzeit also in $\mathcal{O}(n \cdot G(n))$ (bzw. $\mathcal{O}(n \cdot \alpha(n,n))$).

### Test der Automaten aus Abschnitt 1.2.3

Nach MAKESET-Operationen: $\{a\}, \{b\}, \ldots, \{v\}, \ldots, \{z\}$

- $S : (a, v)$

  $\text{FIND}(a) = \{a\} \neq \{v\} = \text{FIND}(v)$

  $\xrightarrow{\text{UNION}} \{a, v\}, \{b\}, \{c\}, \ldots, \{w\}, \{x\}, \{y\}, \{z\}$

- $S : (b, w), (f, z)$

  $\text{FIND}(f) = \{f\} \neq \{z\} = \text{FIND}(z)$

  $\xrightarrow{\text{UNION}} \{a, v\}, \{b\}, \{c\}, \ldots, \{f, z\}, \ldots, \{w\}, \{x\}, \{y\}$

- $S : (b, w), (c, y), (g, x)$

  $\text{FIND}(g) = \{g\} \neq \{x\} = \text{FIND}(x)$

  $\xrightarrow{\text{UNION}} \{a, v\}, \{b\}, \{c\}, \ldots, \{f, z\}, \{g, x\} \ldots, \{w\}, \{y\}$

- $S : (b, w), (c, y), (g, x), (e, v)$

  $\text{FIND}(e) = \{e\} \neq \{a, v\} = \text{FIND}(v)$

  $\xrightarrow{\text{UNION}} \{a, e, v\}, \{b\}, \{c\}, \{d\}, \{f, z\}, \{g, x\}, \{h\}, \{w\}, \{y\}$

- $S : (b, w), (c, y), (g, x), (h, w), (f, z)$

$\textsc{Find}(f) = \{f, z\} = \textsc{Find}(z)$

- $S : (b, w), (c, y), (g, x), (h, w)$

  $\textsc{Find}(h) = \{h\} \neq \{w\} = \textsc{Find}(w)$

  $\xrightarrow{\textsc{Union}} \{a, e, v\}, \{b\}, \{c\}, \{d\}, \{f, z\}, \{g, x\}, \{h, w\}, \{y\}$

- $S : (b, w), (c, y), (g, x), (g, x), (c, y)$

  $\textsc{Find}(c) = \{c\} \neq \{y\} = \textsc{Find}(y)$

  $\xrightarrow{\textsc{Union}} \{a, e, v\}, \{b\}, \{c, y\}, \{d\}, \{f, z\}, \{g, x\}, \{h, w\}$

- $S : (b, w), (c, y), (g, x), (g, x), (a, v), (c, y)$

- $\vdots$

- $S : (b, w)$

  $\textsc{Find}(b) = \{b\} \neq \{h, w\} = \textsc{Find}(w)$

  $\xrightarrow{\textsc{Union}} \{a, e, v\}, \{b, h, w\}, \{c, y\}, \{d\}, \{f, z\}, \{g, x\}$

- $S = \emptyset$, Mengen $\{a, e, v\}, \{b, h, w\}, \{c, y\}, \{d\}, \{f, z\}, \{g, x\}$.

Die Endzustände $c$ und $y$ sind nicht mit Nicht-Endzuständen zusammen in einer Menge, also ist $\mathcal{A}_1 \equiv \mathcal{A}_2$.

## 1.3 Priority Queues oder Heaps

Gesucht ist eine Datenstruktur $H$ um eine geordnete Menge $M$ zu verwalten, die folgende Operationen unterstützt:

- Findmax: gibt den maximalen Wert an, der in $H$ abgelegt ist (in $\Theta(1)$)
- Delete$(H, i)$: entfernt das Element an der Stelle $i$ in $H$ (in $O(\log n)$)
- Insert$(H, x)$: fügt einen neuen Wert $x$ in $H$ ein (in $\Theta(\log n)$)
- Makeheap$(M)$: bildet die Datenstruktur mit Werten aus $M$ (in $\Theta(n)$)

Eine solche Datenstruktur wird Priority Queue genannt und kann als Heap realisiert werden.

**Definition 1.17.** *Ein* Heap *ist ein voller binärer Baum, der mit einem Array A realisiert wird. Die Indizierung von A wird entsprechend der Indizierung der Knoten von der Wurzel nach unten und im gleichen Level von links nach rechts angelegt. Der* Heap *erfüllt zusätzlich die* Heap*-Eigenschaft, d.h.:*

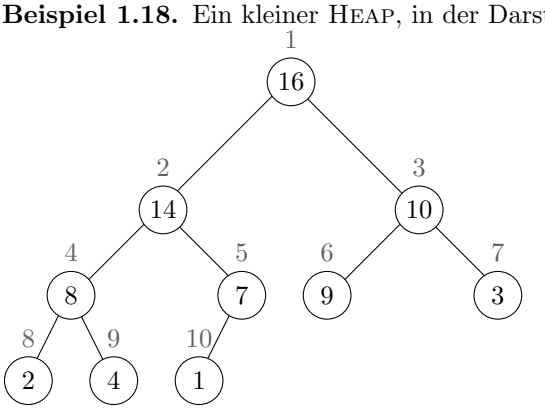
$$\forall i : \quad A[\mathit{Vorgänger}[i]] \geq A[i]$$

*oder äquivalent dazu*

$$A[i] \geq A[2i] \text{ und } A[i] \geq A[2i+1].$$

*Im Folgenden nutzen wir letztere Formulierung der* Heap*-Eigenschaft für ein Element* $A[i]$.

**Beispiel 1.18.** Ein kleiner Heap, in der Darstellung als Baum und als Array.




$$A = \begin{cases} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 16 & 14 & 10 & 8 & 7 & 9 & 3 & 2 & 4 & 1 \end{cases}$$ ∎

**Bemerkung 1.19.** *Jeder Unterbaum eines* Heap *ist wieder ein* Heap.

### 1.3.1 Heapify

Heapify ist eine Prozedur zur Aufrechterhaltung der Heap-Eigenschaft.
Annahme: Für die beiden Unterbäume mit Wurzel $2i$ bzw. $2i+1$ sei jeweils die Heap-Eigenschaft erfüllt, aber für $i$ gelte:

$$A[i] < A[2i] \quad \text{oder} \quad A[i] < A[2i+1]$$

Sei weiterhin Heap-Größe($A$) die Anzahl der Werte im Heap.

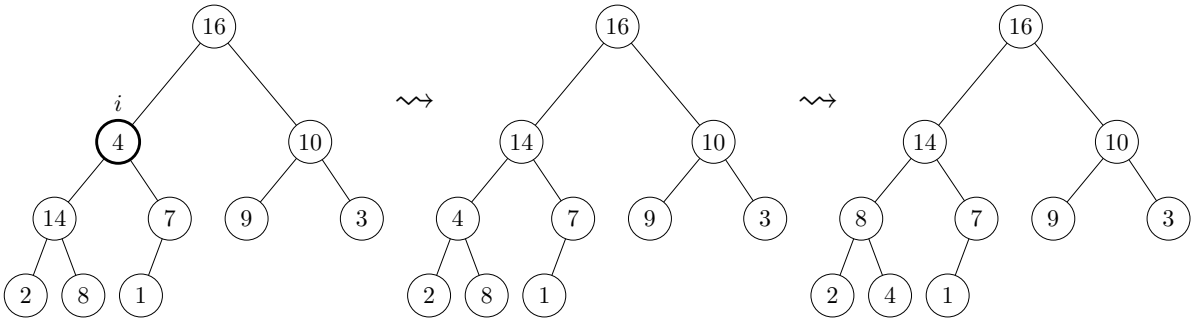



Abbildung 1.10: Beispiel: Heapify tauscht den $i$-ten Knoten, der die Heap-Bedingung bricht, nach unten.

Korrektheit: ist klar.
Laufzeit: Sei $h(i)$ definiert als die Höhe des Unterbaumes mit Wurzel $i$ und sei weiterhin Heap-Größe($A$) $= n$. Dann ist $h(i) \in O(\log n)$ und somit die Rekursionstiefe in $O(\log n)$. Dann folgt für die Laufzeit $T(n)$ für Heapify($A, i$): $T(n) \in O(\log n)$

**Bemerkung 1.20.** *Man betrachte einen vollen binären Baum mit n Knoten. Im Unterbaum mit Höhe h von Knoten v sind* $|T(v)|$ *Knoten. Beachte dass* $T(v)$ *in diesem Zusammenhang keine Laufzeit angibt, sondern einen* Baum *(nach englisch:* Tree*). Es sollte im Folgenden aus dem Zusammenhang hervorgehen, welche Bedeutung T jeweils hat. Der Anteil dieser Knoten, welcher sich*

**Algorithmus 19** : HEAPIFY$(A, i)$

**Eingabe** : Vollst. binärer Baum als Array $A$, Index $i$
**Ausgabe** : Das Array $A$, im Unterbaum von $i$ als HEAP
**Vorbedingungen** : Unterbäume der Wurzeln $A[2i]$ und $A[2i+1]$ sind bereits ein HEAP

```
Wenn 2i ≤ HEAP-Größe(A) und A[2i] > A[i]
    Max-Index ← 2i
sonst
    Max-Index ← i
Wenn 2i + 1 ≤ HEAP-Größe(A) und A[2i + 1] > A[Max-Index]
    Max-Index ← 2i + 1
Wenn Max-Index ≠ i
    tausche A[i] und A[Max-Index]
    rufe rekursiv HEAPIFY(A,Max-Index) auf
```

*im Unterbaum des linken Nachfolgers $l(v)$ von $v$ befindet ist:*

$$\frac{|T(l(v))|}{|T(v)|} = \frac{|T(l(v))|}{|T(l(v))| + |T(r(v))| + 1} \leq \frac{2^h - 1}{2^h - 1 + 2^{h-1} - 1 + 1} \leq \frac{2^h}{2^h + 2^{h-1}} = \frac{2}{3}$$

*Das erste Ungleichungszeichen gilt, da wir den Anteil der Knoten des linken Nachfolgers nach oben abschätzen und somit den Anteil des rechten Nachfolgers möglichst klein annehmen. Beachte, dass das zweite Ungleichungszeichen gilt, da für $\frac{q}{n} < 1$ stets $\frac{q}{n} < \frac{q+\varepsilon}{n+\varepsilon}, \varepsilon \in \mathbb{R}_+$ gilt, da wir nach oben abschätzen, maximieren wir also $|T(l(v))|$. Im zweiten Schritt der Abschätzung gilt Ähnliches. Es folgt, dass in einem vollen binären Baum mit $n$ Knoten für jeden Knoten $v$ gilt:*

$$|T(Nachfolger(v))| \leq 2/3 \cdot |T(v)| \leq 2/3 \cdot n$$

*Eine alternative Analyse der Laufzeit von Algorithmus 19 ergibt also:*

$$T(n) \leq T\left(\frac{2}{3}n\right) + c \quad \textit{mit Konstante } c \ .$$

*Mit Hilfe des Master-Theorems 0.14 gilt:*

$$T(n) \in \Theta(n^{\log_b a} \cdot \log n) \text{ wobei } \log_b a = log_{3/2} 1 = 0, \text{ d.h. } T(n) \in \Theta(\log n)$$

### 1.3.2 Makeheap$(M)$

Sei $|M| = n$ und zu Beginn seien die Elemente in $M$ in beliebiger Reihenfolge in $A[1], \ldots, A[n]$ abgelegt. Betrachte nun Algorithmus 20 zur Durchsetzung der HEAP-Eigenschaft in $A$.
Beachte: In $A[\lfloor n/2 \rfloor + 1, \ldots, n]$ ist die HEAP-Eigenschaft erfüllt, da die entsprechenden Knoten alles Blätter sind.

**Algorithmus 20** : MAKEHEAP $(M)$

**Eingabe** : Vollst. binärer Baum als Array $A$
**Ausgabe** : Das Array $A$ als HEAP

```
Für i = ⌊n/2⌋, ..., 1
    HEAPIFY(A, i)
```

Korrektheit: Die Korrektheit folgt aus der Korrektheit von HEAPIFY und Reihenfolge der Aufrufe.
Laufzeit: Es erfolgen $\lfloor n/2 \rfloor$ Aufrufe von HEAPIFY, wobei die Laufzeit von HEAPIFY im Worst-case in $\Theta(\log n)$ liegt. Somit ergibt sich eine Gesamtlaufzeit von $O(n \log n)$. Doch diese Abschätzung ist zu grob. Eine genauere Betrachtung ergibt Folgendes:

- Die Laufzeit von HEAPIFY$(A, i)$, wobei $i$ im $j$-ten Level des Baumes liegt, ist in $\Theta(\log(n) - j)$
- Die Gesamtlaufzeit $T(n)$ von MAKEHEAP ist dann:

$$
\begin{aligned}
T(n) &\leq \underbrace{\sum_{j=0}^{\lceil \log n \rceil - 1}}_{\text{Alle Level, bis auf unterstes}} \underbrace{2^j}_{\text{Max. Anzahl Knoten in Level } j} \cdot \underbrace{c \cdot (\lceil \log n \rceil - j)}_{\text{Laufzeit pro HEAPIFY in Level } j} \\
&= \sum_{j=1}^{\lceil \log n \rceil} 2^{\lceil \log n \rceil - j} \cdot c \cdot j \qquad \text{Umkehrung der Summationsreihenfolge} \\
&= c \cdot \underbrace{2^{\lceil \log n \rceil}}_{\leq 2^{1+log n} = 2n} \underbrace{\sum_{j=0}^{\lceil \log n \rceil} \frac{j}{2^j}}_{< 2 \text{ (s. Bem. 1.21)}} \\
&\leq c \cdot 2 \cdot (2n) \in \Theta(n)
\end{aligned}
$$

**Bemerkung 1.21.** *Für die Reihe* $\sum_{j=0}^{\infty} \frac{j}{2^j}$ *gilt:*

$$
\begin{aligned}
\limsup_{j \to \infty} \left| \frac{a_{j+1}}{a_{ji}} \right| &= \limsup_{j \to \infty} \frac{(j+1)/2^{j+1}}{j/2^j} = \limsup_{j \to \infty} \frac{(j+1)}{2j} = \frac{1}{2} < 1 \quad \Rightarrow \textit{absolut konvergent} \\
\sum_{j=0}^{\infty} \frac{j}{2^j} &= \sum_{j=1}^{\infty} \frac{j}{2^j} = \sum_{j=1}^{\infty} j \cdot \underbrace{\left( \frac{1}{2^{j-1}} - \frac{1}{2^j} \right)}_{= \frac{1}{2^j}} \\
&= \Big( \underbrace{\frac{1}{2^0}}_{=\frac{1}{2^0}} \underbrace{- \frac{1}{2^1}}_{=\frac{1}{2^1}} \Big) + \Big( \underbrace{\frac{2}{2^1}}_{=\frac{1}{2^1}} \underbrace{- \frac{2}{2^2}}_{=\frac{1}{2^2}} \Big) + \Big( \underbrace{\frac{3}{2^2}}_{=\frac{1}{2^2}} \underbrace{- \frac{3}{2^3}}_{=\frac{1}{2^3}} \Big) + \Big( \underbrace{\frac{4}{2^3}}_{=\frac{1}{2^3}} \underbrace{- \frac{4}{2^4}}_{=\frac{1}{2^4}} \Big) + \Big( \frac{5}{2^4} \quad \dots \Big) \dots \\
&= \sum_{j=0}^{\infty} \frac{1}{2^j} = 2
\end{aligned}
$$

### 1.3.3 Prozedur Delete $(A, i)$

**Algorithmus 21** : DELETE $(A, i)$

**Eingabe** : HEAP der Größe $n$, zu löschendes Element $i$
**Ausgabe** : Das Array $A \setminus i$ als Heap

```
1 A[i] ← A[n]
2 n ← n − 1
3 Wenn A[i] ≤ A[⌊i/2⌋]
4   HEAPIFY (A, i)
5 sonst
6   SIFT-UP (A, i)
```

Die Prozedur DELETE $(A, i)$ entfernt den Wert, der an der Stelle $i$ in $A$ steht und fügt dort stattdessen den Wert ein, der an der Stelle $n$ steht. Dann wird $A[n]$ implizit entfernt indem die Größe des HEAP um eins reduziert wird. Was geschieht bei DELETE mit der HEAP-Eigenschaft? Betrachte das Element $A[i]$ nach eine Ausführung von DELETE $(A, i)$:
Fall 1: Es gelte $A[i] \leq A[\lfloor i/2 \rfloor]$, d.h. die HEAP-Eigenschaft ist oberhalb von $i$ erfüllt. Daher wird nun HEAPIFY$(A, i)$ aufgerufen, um die HEAP-Eigenschaft im Unterbaum von $i$ zu gewährleisten. Die Laufzeit dafür ist in $O(\log n)$.

**Bemerkung 1.22.** *Der oft benötige Spezialfall* DELETE$(A, 1)$ = DELETEMAX *kann so zusammen mit dem Aufruf* HEAPIFY$(A, 1)$ *in* $O(\log n)$ *ausgeführt werden.*

Fall 2: Es gelte $A[i] > A[\lfloor i/2 \rfloor]$, d.h. die HEAP-Eigenschaft ist sicher im Unterbaum von $i$ erfüllt aber nicht in $A[\lfloor i/2 \rfloor]$. Die HEAP-Eigenschaft könnte in $A[\lfloor i/2 \rfloor]$ mit Hilfe von HEAPIFY wiederhergestellt werden, wäre dann aber möglicherweise wieder im Vorgänger von $A[\lfloor i/2 \rfloor]$ verletzt. Wir wissen allerdings, dass HEAPIFY$(A, \lfloor i/2 \rfloor)$ nur einmal aufgerufen würde (nicht rekursiv), da $A[\lfloor i/2 \rfloor]$ vor DELETE$(A, i)$ das Maximum des Unterbaums von $\lfloor i/2 \rfloor$ ist. Es muss also ähnlich wie bei HEAPIFY nun nach *oben* getauscht werden, dazu ist die Prozedur SIFT-UP$(A, i)$ im folgenden Abschnitt geeignet.

### 1.3.4 Prozedur Sift-Up$(A, i)$

Die Prozedur SIFT-UP$(A, i)$ verschiebt ein Element durch Tauschen mit dem Vorgänger solange im Baum nach oben, bis in seinem Vorgänger die HEAP-Bedingung erfüllt ist.

**Algorithmus 22** : SIFT-UP$(A, i)$

**Eingabe** : Vollst. binärer Baum als Array $A$, HEAP-Eigenschaft erfüllt, bis auf evtl. in $i$
**Ausgabe** : Das Array $A$ als HEAP

```
1 ℓ ← i
2 Solange ⌊ℓ/2⌋ > 0 und A[ℓ] > A[⌊ℓ/2⌋] tue
3 |  Vertausche A[ℓ] und A[⌊ℓ/2⌋]
4 |  ℓ ← ⌊ℓ/2⌋
```

Korrektheit: Die Prozedur SIFT-UP$(A, i)$ erfüllt die Invariante, dass nach jedem Tausch von $A[\ell]$ und $A[\lfloor \ell/2 \rfloor]$ die HEAP-Eigenschaft im Unterbaum von $A[\lfloor \ell/2 \rfloor]$ garantiert ist.
Laufzeit: Die Laufzeit von SIFT-UP$(A, i)$ ist in $O(\log n)$, bedingt durch die Baumtiefe.

### 1.3.5 Prozedur Insert $(A, x)$

Ein Element wird in einen HEAP eingefügt, indem es zunächst an das Ende des Arrays angehängt wird und dann mit Hilfe der Prozedur SIFT-UP solange nach oben getauscht wird, bis es einen Platz einnimmt, an dem es die HEAP-Eigenschaft seines Vorgängers erfüllt.

**Algorithmus 23** : INSERT$(A, x)$

**Eingabe** : HEAP, einzufügender Wert $x$
**Ausgabe** : HEAP inklusive $x$

```
1 HEAP-Größe(A) ← HEAP-Größe(A) + 1
2 Füge x in A[n + 1] ein
3 SIFT-UP(A, n + 1)
```

Korrektheit: Die Korrektheit folgt aus der Korrektheit von Sift-Up.
Laufzeit: Die Laufzeit von Insert ist in $O(\log n)$.

### 1.3.6 Sortierverfahren Heapsort$(M)$

Elemente aus der Menge $M$ seien in einem Array $A$ der Länge $n$ abgelegt.

**Algorithmus 24** : Heapsort$(A)$

**Eingabe** : Array $A$ der Länge $n$
**Ausgabe** : Aufsteigend sortiertes Array $A$

```
1 Makeheap(A)
2 Für i = n, ..., 2
3 |  Vertausche A[1] und A[i]
4 |  Heap-Größe(A) ← Heap-Größe(A) − 1
5 |  Heapify(A, 1)
```

Korrektheit: Die Schleife erfüllt die Invariante, dass jeweils $A[i-1,\ldots,n]$ aufsteigend sortiert ist.
Laufzeit: Die Laufzeit von Heapsort ist $T(n) = c_1 n + (n-2)(c_2 + c_3 \log n) \in O(n \log n)$.

Eine genauere Analyse der Anzahl Vergleiche liefert Folgendes:
Heapify führt pro Level, das durchlaufen wird, 2 Vergleiche aus. Etwa die Hälfte aller Knoten in einem vollen binären Baum sind Blätter, etwa ein Viertel aller Knoten haben nur Blätter als Nachfolger etc. Somit folgt, dass bei der Hälfte der Aufrufe Heapify$(A,1)$ ein Element bis zu einem Blatt absinkt, in 3/4 der Fälle „fast“ bis zu einem Blatt. Also finden im Durchschnitt $2 \cdot n \cdot \log n$ Vergleiche statt (nach Makeheap).

### 1.3.7 Alternative Bottom-Up-Heapify $(A, 1)$

Die Prozedur Bottom-Up-Heapify$(A,1)$ stellt eine alternative Technik dar, die Heap-Struktur durchzusetzen, nachdem in $A[1]$ ein neues Element eingefügt wurde, wie zum Beispiel bei Heapsort (Algorithmus 24, Zeilen 3 bzw. 5). Bottom-Up-Heapify spart gegenüber Heapify einige Vergleiche ein, indem es ausnutzt, dass ein eingefügtes Element im Baum recht weit nach unten getauscht werden muss. Die Prozedur bestimmt den Zielpfad entlang dem $A[1]$ absinken würde, indem immer der größere der beiden Nachfolger eines Knotens als nächster Knoten im Pfad aufgenommen wird (Zeilen 1 bis 6). Dann wird vom Blatt des Pfads aus mit dem Element $A[i]$ nach oben gewandert, bis die richtige Position erreicht ist (Zeilen 7 bis 8), das Element dort eingefügt und anschließend alle anderen Elemente bis zur Wurzel um eine Stufe nach oben geschoben (Zeilen 9 bis 14).

Die Anzahl der Vergleiche bei Bottom-Up-Heapify$(A,1)$ ist anstatt $2 \cdot \log n$ nur $1 \cdot \log n + \varepsilon$, wobei im Mittel $\varepsilon \leq 2$. Beachte dazu, dass etwa die Hälfte aller Knoten in einem Baum in den Blättern liegen, ein Viertel wiederum eine Stufe darüber usw. Daraus ergibt sich für die mittlere Anzahl $\varepsilon$ an Aufrufen von Zeile 7:

$$\varepsilon \leq \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

Allerdings entsteht zusätzlicher Aufwand für das „Hochschieben“, um die Wurzel wieder zu besetzen. Dies kann jedoch schnell implementiert werden, da keine Vergleiche mehr stattfinden.

**Algorithmus 25** : BOTTOM-UP-HEAPIFY$(A, 1)$

**Eingabe** : Array $A$, bis auf in $A[1]$ als HEAP
**Ausgabe** : Array $A$ als HEAP

```
1  j ← 1
2  Solange 2j < Heap-Größe tue
3  |  Wenn A[2j] ≥ A[2j + 1]
4  |  |  j ← 2j
5  |  sonst
6  |  |  j ← 2j + 1
7  Solange A[1] ≥ A[j] tue
8  |  j ← ⌊j/2⌋
9  k ← A[j]
10 A[j] ← A[1]
11 j ← ⌊j/2⌋
12 Solange j > 0 tue
13 |  Tausche k und A[j]
14 |  j ← ⌊j/2⌋
```

# Kapitel 2

# Aufspannende Bäume minimalen Gewichts

**Der Literaturtip.** Der „Algorithmus von Kruskal“ ist in [14] beschrieben. Bereits 1930 wurde von Jarník ein Algorithmus veröffentlicht (in tschechisch), der dem „Algorithmus von Prim“ entspricht. Später ist er unabhängig voneinander von Prim [18] und Dijkstra [5] wiederentdeckt worden. Die Färbungsmethode wird in [20] von Tarjan beschrieben. Dort werden auch andere Varianten der Färbungsmethode angegeben. Folgen von UNION– und FIND–Operationen und Datenstrukturen vom Typ HEAP, sowie die effiziente Implementationen der Algorithmen von Kruskal und Prim unter Benutzung dieser Konzepte sind ebenfalls genauer in [20] beschrieben. Matroide und deren Zusammenhang mit aufspannenden Bäumen sind etwa in [12] zu finden.

## 2.1 Einführung

Wir benutzen folgende Grundbegriffe der Graphentheorie. Bezeichne das Paar $G = (V, E)$ einen *ungerichteten Graphen* mit endlicher *Knotenmenge* $V$ und *Kantenmenge* $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. Ein *Weg* in $G$ ist eine Folge $v_1, v_2, \ldots, v_k$ von Knoten aus $V$, in der zwei aufeinanderfolgende Knoten durch eine Kante aus $E$ verbunden sind. Ein Graph $G = (V, E)$ heißt *zusammenhängend*, wenn es zwischen je zwei Knoten $u, v \in V$ einen Weg in $G$ gibt. Ein Pfad ist ein Weg in dem jeder Knoten nur einmal auftritt. Ein zusammenhängender Graph $B = (V(B), E(B))$ heißt *Baum*, wenn es zwischen je zwei Knoten aus $V(B)$ *genau einen* Pfad in $B$ gibt. Ein zusammenhängender Teilgraph $B = (V(B), E(B))$ von $G = (V, E)$, $E(B) \subseteq E$ heißt *aufspannend*, wenn $V(B) = V$.

## 2.2 Das MST-Problem

**Definition 2.1 (Das MST–Problem).** [1]
*Gegeben sei ein zusammenhängender Graph $G = (V, E)$ und eine Gewichtsfunktion $c : E \longrightarrow \mathbb{R}$. Finde einen aufspannenden Teilgraphen $B = (V, E')$ von $G$, mit $E' \subseteq E$, der ein Baum ist und bezüglich $c$ minimales Gewicht hat. Das heißt so, dass*

$$c(B) = \sum_{\{u,v\} \in E'} c(\{u, v\})$$

*minimal über alle aufspannenden Bäume in $G$ ist.*

[1] MST steht für **M**inimum **S**panning **T**ree

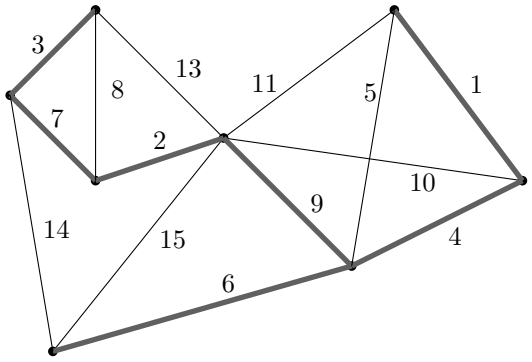



Abbildung 2.1: Ein aufspannender Baum minimalen Gewichts

### 2.2.1 Motivation

Das MST–Problem ist ein Grundproblem der algorithmischen Graphentheorie, das viele Anwendungen hat, etwa beim Entwurf eines Netzwerkes, das geographisch verteilte Komponenten möglichst günstig miteinander verbinden soll, um beispielsweise Kommunikationsmöglichkeiten oder Infrastruktur zur Verfügung zu stellen.

Wir wollen effiziente Algorithmen zur Lösung des MST–Problems entwerfen. Alle aufspannenden Bäume in einem Graphen zu ermitteln und einen kostenminimalen daraus auszuwählen ist sicher keine „effiziente“ Vorgehensweise: Man kann unter Benutzung der sogenannten „Prüfer–Korrespondenz“ beweisen, dass es in einem vollständigen Graphen mit $n$ Knoten $n^{n-2}$ aufspannende Bäume gibt (dies ist der Satz von Cayley.) Dazu zeigt man, daß es eine bijektive Abbildung zwischen der Menge aller aufspannenden Bäume über $n$ Knoten und der Menge aller Wörter der Länge $n-2$ über dem Alphabet $\{1,\ldots,n\}$ gibt.

Im nächsten Abschnitt formulieren wir eine allgemeine Vorgehensweise zur Lösung des MST–Problems, die alle bisher bekannten Algorithmen für das MST–Problem verallgemeinert. Diese allgemeine Methode, genannt „Färbungsmethode“, ist von R. E. Tarjan eingeführt worden. Wir werden sehen, daß zwei klassische Algorithmen zur Lösung des MST–Problems, der „Algorithmus von Kruskal“ und der „Algorithmus von Prim“, nur spezielle Varianten der Färbungsmethode sind. Für diese beiden Algorithmen werden wir effiziente Implementationen skizzieren. Die Färbungsmethode kann als ein „Greedy–Verfahren“ im allgemeinen Sinne aufgefaßt werden. Es werden auf der Basis der bisher konstruierten Teillösung Kanten in die Lösung aufgenommen oder aus der Lösung ausgeschlossen. Diese Entscheidungen werden nachträglich nicht mehr rückgängig gemacht. Die Optimalität von Greedy–Verfahren basiert auf einer kombinatorischen Struktur, die im letzten Abschnitt kurz behandelt wird.

## 2.3 Die Färbungsmethode von Tarjan

Die Färbungsmethode färbt Kanten nacheinander *grün* oder *rot*. Am Ende bildet die Menge der grünen Kanten einen aufspannenden Baum minimalen Gewichts. Die Färbungen der Kanten sind Anwendungen von Regeln, einer *grünen Regel* oder einer *roten Regel.* Um diese Regeln zu formulieren, benötigen wir die Begriffe „Schnitt“ und „Kreis“ in einem Graphen.

**Definition 2.2 (Schnitt).** *Ein* Schnitt *in einem Graphen $G=(V,E)$ ist eine Partition $(S, V\setminus S)$ der Knotenmenge $V$. Eine Kante $\{u,v\}$* kreuzt *den Schnitt $(S, V\setminus S)$, falls $u\in S$ und $v\in V\setminus S$ ist. Oft wird auch die Menge der Kanten, die den Schnitt $(S, V\setminus S)$ kreuzt, mit diesem Schnitt identifiziert.*

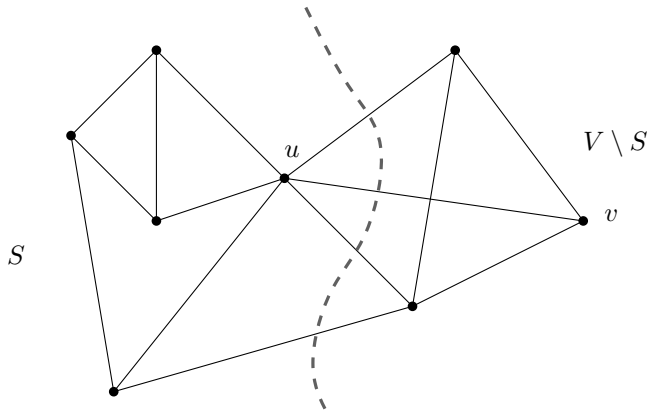



Abbildung 2.2: Beispiel eines Schnitts $(S, V \setminus S)$

**Definition 2.3.** *Ein* Kreis *in einem Graphen $G = (V, E)$ ist eine Folge $v_1, \ldots, v_k = v_1, k > 3$, von Knoten aus $G$, in der zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind und kein Knoten außer dem Anfangs- und Endknoten zweimal auftritt.*

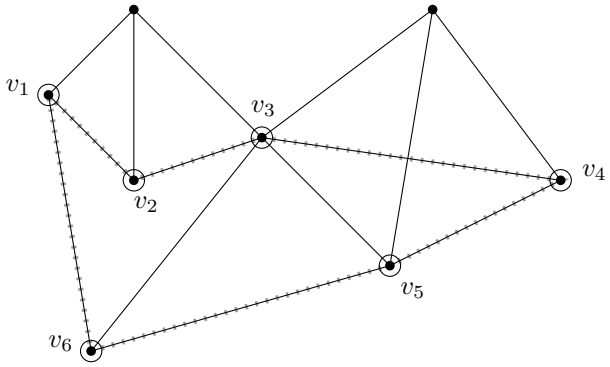



Abbildung 2.3: Die Folge $v_1, v_2, v_3, v_4, v_5, v_6, v_1$ ist ein Kreis.

### 2.3.1 Grüne Regel

Wähle einen *Schnitt* in $G$, der von keiner grünen Kante gekreuzt wird. Unter allen ungefärbten Kanten, die diesen Schnitt kreuzen, wähle eine Kante *minimalen Gewichts* und färbe sie grün.

**Beispiel 2.4.** Abb. 2.4 zeigt eine Anwendung der grünen Regel: Die gestrichelten Kanten sind bereits rot, die breiten Kanten grün gefärbt. Die Kante mit Gewicht 6 wird grün gefärbt.

### 2.3.2 Rote Regel.

Wähle einen *Kreis*, der keine rote Kante enthält. Unter allen ungefärbten Kanten, die auf diesem Kreis liegen, wähle eine Kante *maximalen Gewichts* und färbe sie rot.

**Beispiel 2.5.** Abb. 2.5 zeigt eine Anwendung der roten Regel: Wieder sind die gestrichelten Kanten bereits rot, die breiten Kanten grün gefärbt. Wir wählen den Kreis aus obigem Beispiel. Die Kante mit Gewicht 14 wird rot gefärbt.

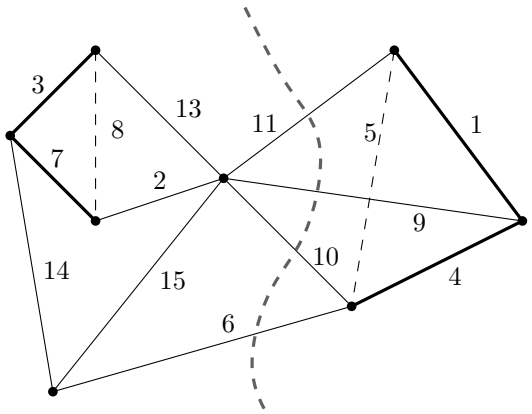



Abbildung 2.4: Eine Anwendung der grünen Regel

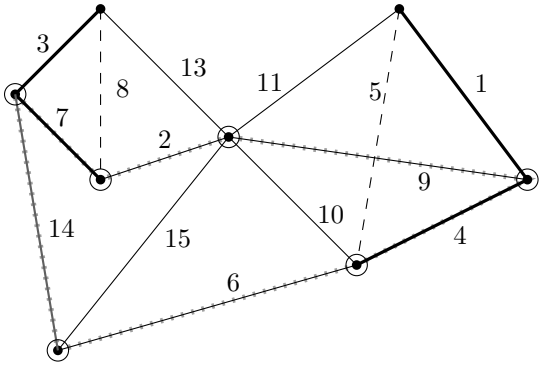



Abbildung 2.5: Eine Anwendung der roten Regel

**Algorithmus 26** : Färbungsmethode von Tarjan

**Eingabe** : Graph mit gewichteten Kanten
**Ausgabe** : Aufspannender Baum minimalen Gewichts in Form der grünen Kanten
1 **Solange** *noch eine der beiden Regeln anwendbar* **tue**
2 Wende die grüne oder die rote Regel an

Die Färbungsmethode ist in Algorithmus 26 formal formuliert. Sie ist nichtdeterministisch. Im allgemeinen gibt es in einem Schleifendurchlauf verschiedene Wahlmöglichkeiten. Einerseits kann die Regel, welche angewandt wird, gewählt werden, andererseits die Kante, auf die die gewählte Regel angewandt wird. Die Behauptung ist, dass am Ende die Menge aller grün gefärbten Kanten einen aufspannenden Baum minimalen Gewichts in $G$ induziert. Um die Korrektheit des Verfahrens zu beweisen formulieren wir die folgende „Invariante“ für die Färbungsmethode.

### 2.3.3 Färbungsinvariante

Es gibt einen aufspannenden Baum minimalen Gewichts, der *alle grünen* Kanten und *keine rote* Kante enthält.

Wir werden beweisen, daß die Färbungsmethode die Färbungsinvariante erhält. Ist erst dann keine der beiden Regeln mehr anwendbar, wenn alle Kanten gefärbt sind, so folgt die Korrektheit des

Färbungsalgorithmus aus der Färbungsinvariante.

**Satz 2.6 (Satz über die Färbungsinvariante).** *Die Färbungsmethode, angewandt auf einen zusammenhängenden Graphen, erhält die Färbungsinvariante. Nach jedem Färbungsschritt gibt es also einen aufspannenden Baum minimalen Gewichts, der alle grünen Kanten und keine rote Kante enthält.*

*Beweis.* Wir beweisen den Satz über die Färbungsinvariante durch eine Induktion über die Anzahl $m$ der Färbungsschritte.

**Induktionsanfang ($m = 0$).** Alle Kanten sind ungefärbt und jeder aufspannende Baum minimalen Gewichts erfüllt die Färbungsinvariante. Da der Graph zusammenhängend ist, existiert mindestens ein aufspannender Baum.

**Induktionsschluss ($m \longrightarrow m + 1$).** Für den $(m + 1)$–ten Färbungsschritt sind zwei Fälle zu unterscheiden. Er ist entweder eine Anwendung der grünen Regel oder eine Anwendung der roten Regel. Wir betrachten die Kante $e$, auf die der $(m + 1)$–te Färbungsschritt angewandt wurde.

**Fall 1**: Der $(m+1)$–te Färbungsschritt ist eine Anwendung der grünen Regel auf die Kante $e$. Nach Induktionsvoraussetzung existiert nach dem $m$–ten Färbungsschritt ein aufspannender Baum $B$ minimalen Gewichts, der alle grünen Kanten und keine rote Kante enthält. Ist $e$ in $B$ enthalten, so erfüllt $B$ auch nach dem $(m + 1)$–ten Färbungsschritt diese Bedingung.

Ist $e$ nicht in $B$ enthalten, so betrachte den Schnitt, auf den die grüne Regel im $(m + 1)$–ten Färbungsschritt angewandt wurde (vgl. Abb. 2.6). Da $B$ zusammenhängend und aufspannend ist, muss es in $B$ einen Weg geben, der die Endknoten von $e$ enthält und mindestens eine Kante $e'$, die den betrachteten Schnitt kreuzt. Da $B$ keine rote Kante enthält und die grüne Regel auf den Schnitt mit $e$ angewandt wird, ist $e'$ ungefärbt. Wegen der Wahl von $e$ ist $c(e') \geq c(e)$.

Durch Wegnahme von $e'$ und Hinzunahme von $e$ entsteht aus $B$ dann wieder ein Baum $B'$. Dieser Baum $B'$ ist wiederum ein aufspannender Baum minimalen Gewichts, der die Färbungsinvariante erfüllt.

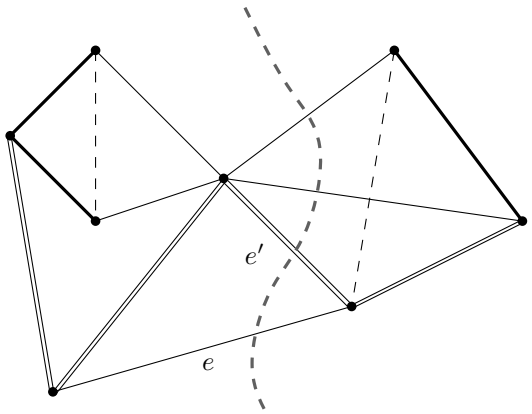



Abbildung 2.6: Die gestrichelten Kanten sind rot, die breiten Kanten grün gefärbt. Die grünen Kanten bilden zusammen mit den doppelt gezeichneten Kanten den Baum $B$. Durch Austausch von $e'$ und $e$ erhalten wir aus $B$ einen aufspannenden Baum $B'$ mit kleinerem Gewicht, der wiederum die Färbungsinvariante erfüllt. Es gilt $B' = B - e' + e$.

**Fall 2:** Der $(m + 1)$–te Färbungsschritt ist eine Anwendung der roten Regel auf die Kante $e$. Sei $B$ wieder der nach Induktionsvoraussetzung nach dem $m$–ten Färbungsschritt existierende aufspannende Baum minimalen Gewichts, der die Färbungsinvariante erfüllt. Falls $e$ nicht in $B$ ist, so erfüllt $B$ auch nach dem $(m + 1)$–ten Färbungsschritt die Färbungsinvariante.

Ist $e$ in $B$ enthalten, so zerfällt $B$ nach Wegnahme von $e$ in zwei Teilbäume, deren Knotenmengen einen Schnitt im Graphen induzieren, der von $e$ gekreuzt wird. Betrachte den Kreis, auf den die rote Regel im $(m+1)$–ten Färbungsschritt angewandt wurde. Auf diesem Kreis liegt eine Kante $e' \neq e$, die ebenfalls den Schnitt kreuzt und nicht rot gefärbt ist. Die Kante $e'$ ist auch nicht grün gefärbt, da nach Definition des Baumes nicht beide, $e$ und $e'$, zu $B$ gehören können. Da die rote Regel im $(m+1)$–ten Färbungsschritt auf $e$ angewandt wird, ist $c(e) \geq c(e')$.

Der Baum $B'$, der aus $B$ durch Wegnahme von $e$ und Hinzunahme von $e'$ entsteht, ist dann wieder ein aufspannender Baum minimalen Gewichts, der die Färbungsinvariante erfüllt. □

**Satz 2.7.** *Die Färbungsmethode färbt alle Kanten eines zusammenhängenden Graphen rot oder grün.*

*Beweis.* Falls die Färbungsmethode endet bevor alle Kanten gefärbt sind, so existiert einerseits eine Kante $e$, die nicht gefärbt ist, andererseits ist weder die rote noch die grüne Regel anwendbar. Da die Färbungsinvariante erfüllt ist, induzieren die grünen Kanten eine Menge von „grünen Bäumen“ (wobei jeder Knoten als „grün“ aufgefasst wird).

**Fall 1.** Beide Endknoten der ungefärbten Kante $e$ liegen in demselben grünen Baum. Dann bildet $e$ zusammen mit dem Weg in diesem Baum, der die Endknoten von $e$ verbindet, einen Kreis, auf den die rote Regel anwendbar ist.

**Fall 2.** Die Endknoten von $e$ liegen in verschiedenen grünen Bäumen. Dann existiert ein Schnitt, der von $e$ gekreuzt wird, und auf den die grüne Regel anwendbar ist. Betrachte dazu einfach einen der Schnitte, die durch die beiden grünen Bäume, in denen die Endknoten von $e$ liegen, induziert wird. □

## 2.4 Der Algorithmus von Kruskal

Der Algorithmus von Kruskal lässt sich nun einfach als eine Variante der Färbungsmethode formulieren. Siehe auch Algorithmus 16 für die formelle Beschreibung des Algorithmus, hier die verbale Beschreibung: Der Algorithmus endet, wenn alle Kanten durchlaufen sind. Der Algorithmus von

**Algorithmus 27** : Algorithmus von Kruskal (verbal)

**Eingabe** : Graph mit gewichteten Kanten
**Ausgabe** : Aufspannender Baum minimalen Gewichts in Form der grünen Kanten

```
1 Sortiere die Kanten nach ihrem Gewicht in nicht–absteigender Reihenfolge
2 Durchlaufe die sortierten Kanten der Reihe nach und wende folgenden Färbungsschritt an
3 Wenn Beide Endknoten der Kante liegen in demselben grünen Baum
4     Färbe sie rot
5 sonst
6     Färbe sie grün
```

Kruskal ist offensichtlich eine spezielle Version der Färbungsmethode, denn jeder Färbungsschritt ist eine Anwendung der grünen oder roten Regel. Betrachte dazu die nächste Kante $e$ in der sortierten Kantenfolge.

**Färbe rot**: Wird die Kante $e$ rot gefärbt, so liegen ihre beiden Endknoten in demselben grünen Baum. Sie schließt also einen Kreis, der keine rote Kante enthält und $e$ als einzige ungefärbte Kante. Damit ist dieser Färbungsschritt eine Anwendung der roten Regel.

**Färbe grün:** Wird die Kante $e$ grün gefärbt, so liegen ihre beiden Endknoten nicht in demselben grünen Baum. Damit induziert sie einen Schnitt, der von keiner anderen grünen Kante gekreuzt wird. Wegen der Sortierung der Kanten ist dieser Färbungsschritt eine Anwendung der grünen Regel.

In einer Implementation des Algorithmus von Kruskal müssen zwei Teilschritte effizient realisiert werden: die Sortierung der Kanten entsprechend ihrem Gewicht und die Organisation der grünen Bäume in einer Form, die den Test „beide Endknoten einer Kante liegen in demselben grünen Baum“ unterstützt. Die Sortierung der Kanten kann in Laufzeit $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$ vorgenommen werden. Die Organisation der (sich verändernden) grünen Bäume wird als Folge von UNION- und FIND-Operationen realisiert:

- FIND: Finde die grünen Bäume, in denen die beiden Endknoten der zu färbenden Kante liegen.
- UNION: Vereinige die beiden grünen Bäume, in denen die beiden Endknoten einer Kante liegen, die grün gefärbt wird.

Wenn eine sortierte Kantenliste gegeben ist, so kann diese Folge von UNION- und FIND-Operationen in Laufzeit $\mathcal{O}(|E| \cdot \alpha(|E|, |V|))$ realisiert werden, wobei $\alpha(|E|, |V|)$ die sehr langsam wachsende Funktion aus Abschnitt 1.1.3. Die Gesamtlaufzeit wird also durch das Sortieren dominiert. In Situationen, in denen die Kanten bereits sortiert vorliegen oder die Kantengewichte so sind, dass sie „schneller“ als in $\mathcal{O}(|E| \log |V|)$ sortiert werden können, ist auch die Gesamtlaufzeit in $\mathcal{O}(|E| \cdot \alpha(|E|, |V|))$.

## 2.5 Der Algorithmus von Prim

Der Algorithmus von Prim lässt sich ebenfalls als spezielle Variante der Färbungsmethode formulieren.

**Algorithmus 28** : Algorithmus von Prim (verbal)

**Eingabe** : Graph $G = (V, E)$
**Ausgabe** : Aufspannender Baum minimalen Gewichts

1 Wähle einen beliebigen Startknoten und betrachte diesen als einen “grünen Baum”
2 **Für** *Färbungsschritt* $i = 1, \ldots, (|V| - 1)$
3 Wähle eine ungefärbte Kante minimalen Gewichts, die genau einen Endknoten in dem grünen Baum hat, und färbe sie grün.

Der Algorithmus von Prim ist offensichtlich eine spezielle Version der Färbungsmethode, denn jeder Färbungsschritt ist eine Anwendung der grünen Regel. Der Schnitt, auf den die grüne Regel angewandt wird, wird jeweils von den Knoten des grünen Baumes induziert, der den Startknoten enthält. Nach genau $|V| - 1$ solcher Färbungsschritte ist nur noch die rote Regel anwendbar, denn zu Beginn können wir die $|V|$ Knoten als $|V|$ disjunkte grüne Bäume auffassen. Mit jedem Färbungsschritt reduziert sich die Anzahl der disjunkten grünen Bäume um genau einen. Nach $|V| - 1$ Färbungsschritten muss also genau ein grüner Baum übrig sein.

### 2.5.1 Implementation des Algorithmus von Prim

Die Unterstützung des Schnitts: „Wähle eine ungefärbte Kante minimalen Gewichts, die genau einen Endknoten in dem grünen Baum hat“ erfolgt mit Hilfe einer geeigneten Datenstruktur. Offensichtlich ist eine Kante $\{u, w\}$ ein Kandidat für „färbe grün“ genau dann, wenn ein Endknoten $u$ oder $w$ im aktuellen grünen Baum $B$ liegt. Wir sagen „der Knoten $u$ begrenzt $B$“ genau dann, wenn $u$ nicht in $B$ ist, aber ein $w \in B$ existiert mit $\{u, w\}$ Kante. Sei nun $H$ ein $d$-HEAP der die folgenden Operationen unterstützt:

- INSERT$(H, x)$ in $O(\log_d n)$,

- DELETEMIN($H$) in $O(d \log_d n)$ (beachte dass $H$ ein min-HEAP ist, sprich umgekehrt sortiert wie ein gewöhnlicher HEAP) und
- DECREASEKEY($H, x, k$), wobei das Element $x$ aus $H$ einen neuen Schlüsselwert $k$ erhält, welcher nicht größer als der alte Schlüsselwert ist, in $O(\log_d n)$.

Datenstrukturen: GRÜN[$v$] enthält

- falls $v \notin B$, aber $v$ begrenzt $B$: Kante minimalen Gewichts inzident zu $v$, deren anderer Endknoten in $B$ liegt,
- falls $v \in B$: Kante, deren „Grünfärbung“ bewirkt hat, dass $v \in B$,
- sonst: undefiniert.

Der grüne Baum $B$ wird induziert durch GRÜN.
KEY[$v$] enthält

- $c(\{v, w\})$ falls $v \notin B$, aber $v$ begrenzt $B$ und $c(\{v, w\})$ ist minimal unter allen Kanten inzident zu $v$, deren anderer Endknoten in $B$ liegt,
- $-\infty$ falls $v \in B$,
- $+\infty$ sonst.

**Algorithmus 29** : Algorithmus von Prim

**Eingabe** : $G(V, E), s \in V$ Startknoten
**Ausgabe** : Kanten $\ell \in E$ für die es $v \in V$ gibt mit GRÜN($v$) $= \ell$

```
1  Für v ∈ V
2      KEY[v] ← ∞
3  v ← s
4  Solange v ist definiert tue
5      KEY[v] ← −∞
6      Für Kanten {v, w} inzident zu v
7          Wenn KEY[w] = ∞
8              KEY[w] ← c({v, w})
9              GRÜN[w] ← {v, w}
10             INSERT(H, w)
11         sonst
12             Wenn c({v, w}) <KEY[w]
13                 KEY[w] ← c({v, w})
14                 GRÜN[w] ← {v, w}
15                 DECREASEKEY(H, w, c({v, w}))
16     v ← DELETEMIN(H)
```

Laufzeit: Sei wie gewohnt $|V| = n$ und $|E| = m$. Jedes $v \in V$ wird nur einmal in Schleife 4. betrachtet und darin jeweils alle zu $v$ inzidenten Kanten. Pro Durchlauf von Schritt 6 wird maximal ein INSERT bzw DECREASEKEY durchgeführt. Somit folgt für die Laufzeit: $T_{\text{Prim}}(n) \in O(n \cdot d \cdot \log_d n + m \cdot \log_d n)$. Falls also gilt: $d := \lceil 2 + m/n \rceil$, so folgt $T_{\text{Prim}}(n) \in O(m \cdot \log_{2+m/n} n)$ und für $m \in \Omega(n^{1+\varepsilon})$ damit $T_{\text{Prim}}(n) \in O(m/\varepsilon)$. Der Algorithmus von Prim ist somit gut geeignet für dichte Graphen, also für Graphen mit $|E| \in \Omega(|V|^{1+\varepsilon}), \varepsilon > 0$. Er ist schlechter als der Algorithmus von Kruskal, wenn die Kanten vorsortiert sind.

## 2.6 Greedy–Verfahren und Matroide

**Definition 2.8.** *Ein Tupel $(M,\mathcal{U})$ wobei $\mathcal{U} \subset 2^M$ ein Mengensystem über einer endlichen Menge $M$ ist, heißt* Unabhängigkeitssystem*, wenn*

- $\emptyset \in \mathcal{U}$ *und*
- $I_1 \in \mathcal{U}, I_2 \subseteq I_1 \Rightarrow I_2 \in \mathcal{U}$.

*Die Mengen $I \subseteq M$ mit $I \in \mathcal{U}$ werden* unabhängig*, alle anderen Mengen $I \subseteq M$* abhängig *genannt.*

Bemerkung: Ein Unabhängigkeitssystem $(M,\mathcal{U})$ bildet mit der Relation $\subseteq$ eine nach unten abgeschlossene Subordnung der durch $\subseteq$ gegeben partiellen Ordnung auf $2^M$ (genannt Filter).

**Definition 2.9.** *Sei $(M,\mathcal{U})$ ein Unabhängigkeitssystem. Für $F \subseteq M$ ist jede unabhängige Menge $U \in \mathcal{U}$, $U \subseteq F$ die bezüglich $\subseteq$ maximal ist eine Basis von $F$, d.h. $B \in \mathcal{U}$ Basis von $F$ genau dann, wenn für $B' \in \mathcal{U}$, $B \subseteq B' \subseteq F$ folgt $B = B'$. Eine Basis von $M$ wird Basis des Unabhängigkeitssystems $(M,\mathcal{U})$ genannt. Die Menge aller Basen von $(M,\mathcal{U})$ heißt* Basissystem von $(M,\mathcal{U})$ *Für $F \subseteq M$ heißt $r(F) := max\{|B| : B$ ist Basis von $F\}$ der* Rang von $F$*. Der Rang von $M$, $r(M)$ wird auch* Rang des Unabhängigkeitssystems *genannt. Eine abhängige Menge, die bezüglich "$\subseteq$" minimal ist, wird auch* Kreis *in $(M,\mathcal{U})$ genannt.*

**Beispiel 2.10.**

- Sei $G = (V, E)$ ein Graph. Alle Teilmengen $E' \subseteq E$, die einen Wald (kreisfreien Graphen) induzieren, bilden ein Unabhängigkeitssystem über $E$ dessen Basen alle aufspannenden Bäume sind und dessen Rang $|V| - 1$ ist.
- Sei $M$ eine endliche Teilmenge eines Vektorraums $V$ und $\mathcal{U} \in 2^M$ so, dass für $X \subseteq M$ genau dann $X \in \mathcal{U}$ gilt, wenn die Vektoren aus $X$ linear unabhängig sind. Das Tupel $(M,\mathcal{U})$ ist ein Unabhängigkeitssystem. Die Rangfunktion von $(M,\mathcal{U})$ ist gerade die Rangfunktion von $V$ reduziert auf den von $M$ aufgespannten Unterraum von $V$. ∎

**Definition 2.11.** *Betrachte ein Unabhängigkeitssystem $(M,\mathcal{U})$ mit einer Gewichtsfuntion $w$ auf $M$. Das Problem eine unabhängige Menge $U^* \in \mathcal{U}$ zu finden, so dass $w(U^*)$ maximal ist, heißt* Optimierungsproblem über dem Unabhängigkeitssystem $(M,\mathcal{U})$*. Sei $\mathcal{B}$ Basissystem. Dann heißt das Problem eine Basis $B^* \in \mathcal{B}$ zu finden, so dass $w(B^*)$ minimal ist ein* Optimierungsproblem über dem Basissystem $\mathcal{B}$*.*

**Beispiel 2.12.** Das MST-Problem ist ein Minimierungsproblem genauer ein Optimierungsproblem über dem Basissystem der „aufspannenden Bäume". ∎

**Algorithmus 30** : Greedy–Methode für ein Optimierungsproblem $\Pi$ über $(M,\mathcal{U})$, $|M| = n$

```
1 Sortiere M aufsteigend (absteigend), falls Π Minimierungsproblem über Basissystem
  (Maximierungsproblem über Unabhängigkeitssystem), die Sortierung sei ℓ_1, ℓ_2, ..., ℓ_n.
2 I* ← ∅
3 Für i = 1, ..., n
4 |   Wenn I* ∪ {ℓ_i} ∈ U
5 |   |   I* ← I* ∪ {ℓ_i}
```

Der Kruskal-Algorithmus (siehe Algorithmen 16 und 27) ist offensichtlich eine Greedy-Methode. Die Effizienz der Greedy-Methode hängt sicher davon ab, wie schnell die Frage "Ist die Menge $I^*$ zusammen mit dem neuen Element unabhängig?" beantwortet werden kann. Dies hängt vom Unabhängigkeitssystem ab. Uns interessiert nun eher die Frage, ob die Greedy-Lösung "gut" ist, beziehungsweise wann sie optimal ist.

**Definition 2.13 (Matroid).** *Ein Unabhängigkeitssystem* $(M,\mathcal{U})$ *heißt* Matroid*, wenn für alle* $I, J \in \mathcal{U}$ *mit* $|I| < |J|$*, ein* $e \in J \setminus I$ *existiert, so dass* $I \cup \{e\} \in \mathcal{U}$*.*

Bemerkung: Äquivalent kann anstatt $|I| < |J|$ auch $|I| + 1 = |J|$ gefordert werden.

**Beispiel 2.14.** Sei $G = (V, E)$ ein zusammenhängender Graph. Das Mengensystem $(E,\mathcal{U})$ mit $\mathcal{U} = \{E' \subseteq E : E'$ induziert einen Wald in $E\}$ ist ein Matroid: Seien $U, W \in \mathcal{U}$ mit $|U| = |W| + 1$. Betrachte alle zusammenhängenden Teilgraphen von $G$, die durch die Kanten induziert werden, die zu $U \cup W$ gehören. Wenn für alle $\{x, y\} \in U \setminus W$ gilt $W \cup \{\{x, y\}\} \notin \mathcal{U}$, dann wird jeder Schnitt in einem der Teilgraphen von einer Kante aus $W$ gekreuzt. $W$ bildet also einen aufspannenden Baum in jedem dieser Teilgraphen und hat damit maximale Kardinalität unter allen unabhängigen Mengen in jedem dieser Teilgraphen, im Widerspruch zu $|U| = |W| + 1$.

**Beispiel 2.15.** Ein Unabhängigkeitssystem, welches kein Matroid ist. Es sei $G = (V, E)$. Das Mengensystem $(V, \mathcal{J})$, wobei

$$\mathcal{J} := \{V' \subseteq V : V' \text{ unabhängige Knotenmenge in } G, \text{ d.h. } \forall u, v \in V' \text{ gilt } \{u, v\} \notin E\}$$ ∎

ist ein Unabhängigkeitsystem, aber kein Matroid (betrachte dazu den Stern in Abbildung 2.7).

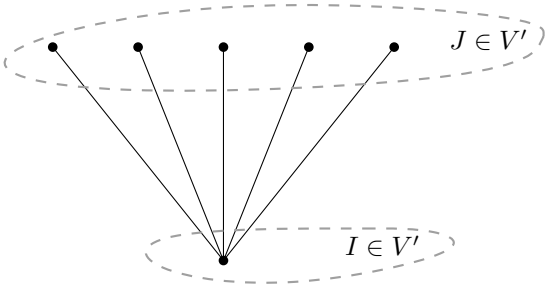



Abbildung 2.7: In diesem Stern gibt es kein $v \in J \setminus I$, so dass $I \cup \{v\} \in \mathcal{J}$.

Folgender Satz klassifiziert die Menge aller Unabhängigkeitssysteme für welche die Greedy-Methode für Optimierungsprobleme über Unabhängigkeitssystemen optimal ist.

**Satz 2.16.** *Für ein Unabhängigkeitssystem* $(M,\mathcal{U})$ *sind folgende Aussagen äquivalent:*

(*a*) *Eine Greedy-Methode liefert eine Optimallösung für das Optimierungsproblem* $\Pi = \max\{w(U) : U \in \mathcal{U}\}$ *über dem Unabhängigkeitssystem* $(M,\mathcal{U})$ *bei beliebiger Gewichtsfunktion* $w$ *über* $M$*.*

(*b*) $(M,\mathcal{U})$ *ist ein Matroid.*

(*c*) *Für eine beliebige Menge* $F \subseteq M$ *und beliebige inklusionsmaximale unabhängige Mengen* $I_1, I_2 \subseteq F$ *gilt:* $|I_1| = |I_2|$*.*

*Beweis.* Wir zeigen $(a) \Rightarrow (b) \Rightarrow (c) \Rightarrow (a)$

$(a) \Rightarrow (b)$ Angenommen die Greedy-Methode für Unabhängigkeitssysteme liefert die Optimallösung für das Problem Π für beliebige Gewichtsfunktion $w$, aber $(M,\mathcal{U})$ ist kein Matroid. Dann existieren Mengen $U, W \in \mathcal{U}$ mit $|U| = |W| + 1$, so dass für alle $e \in U \setminus W$ die Menge $W \cup \{e\} \notin \mathcal{U}$ ist. Betrachte folgende Gewichtsfunktion $w$ auf $M$:

$$w(l) := \begin{cases} |W| + 2 & \text{, falls } e \in W \\ |W| + 1 & \text{, falls } e \in U \setminus W \\ -1 & \text{, falls } e \notin U \cup W \end{cases}$$

Zunächst gilt:

$$w(U) \geq |U|(|W|+1) = (|W|+1)^2 = |W|^2+2|W|+1 > |W|^2+2|W| = |W|(|W|+2) = w(W)$$

Die Greedy-Methode wählt zunächst alle Elemente aus $W$, da diese das größte Gewicht haben. Dann wird sie kein weiteres Element $e \in M$ hinzunehmen, da entweder $W \cup \{e\} \notin \mathcal{U}$ (falls $e \in U \setminus W$) oder $w(e) < 0$. Die Greedy-Methode liefert also als Lösung $W$, obwohl $W$ nicht Optimallösung des Problems $\Pi$ sein kann. Widerspruch.

$(b) \Rightarrow (c)$ Angenommen für $U, W \in \mathcal{U}$ mit $|U| = |W|+1$, existiert ein $e \in U \setminus W$, so dass $W \cup \{e\} \in \mathcal{U}$. Seien $I_1, I_2 \in \mathcal{U}$, $I_1, I_2 \subseteq F \subseteq M$ maximal unabhängig in $F$ und o.B.d.A. $|I_1| < |I_2|$.

Wir können eine Menge $I' \subseteq I_2$ konstruieren, so dass $I' \in \mathcal{U}$ und $|I'| = |I_1| + 1$ ist, indem wir $|I_2| - |I_1| - 1$ Elemente aus $I_2$ streichen, da $\mathcal{U}$ bzgl $\subseteq$ nach unten abgeschlossen ist. Dann existiert $e \in I' \setminus I_1$, so dass $I_1 \cup \{e\} \in \mathcal{U}$ ist und $I_1 \cup \{e\} \subseteq F$. Widerspruch zur Annahme, dass $I_1$ maximal unabhängig in $F$ ist.

$(c) \Rightarrow (a)$ Angenommen alle inklusionsmaximal unabhängigen Mengen einer Teilmenge von $M$ haben gleiche Kardinalität, aber die Greedy-Methode liefert für eine Gewichtsfunktion $w$ keine Optimallösung des Problems $\Pi = \max\{w(U) : U \in \mathcal{U}\}$, d.h. die Greedy-Methode liefert ein $I = \{e_1, \ldots, e_i\} \in \mathcal{U}$ für das ein $J = \{e'_1, \ldots, e'_j\} \in \mathcal{U}$ existiert mit $w(J) > w(I)$. Da $I$ Greedy-Lösung ist, muss $I$ maximal unabhängig in der Menge $F = \{e \in M : w(e) > 0\}$ sein. Wir können davon ausgehen, dass für alle $e \in J$, $w(e) > 0$ gilt, d.h. $J$ ist ebenfalls maximal unabhängig in $F$. Nach Voraussetzung ist also $i = j$.

O.B.d.A Seien die Elemente aus $I$ bzw $J$ folgendermaßen angeordnet:

$$\begin{array}{llll} w(e_1) & \geq w(e_2) & \geq \ldots & \geq w(e_i) \\ w(e'_1) & \geq w(e'_2) & \geq \ldots & \geq w(e'_j) \end{array}$$

Per Induktion über die Indizes der Elemente aus $I$ bzw $J$ zeigen wir $w(e_k) \geq w(e'_k)$. Für $k = 1$ gilt $w(e_1) \geq w(e'_1)$, da die Greedy-Methode $e_1$ als erstes Element wählt. Induktionsschritt $k \to k+1$: Angenommen $w(e_k) < w(e'_k)$. Betrachte die Menge $F' := \{e \in M : w(e) \geq w(e'_k)\}$. Die Menge $\{e_1, \ldots e_{k-1}\}$ ist maximal unabhängig in $F'$, denn falls für $e \in F'$ gilt $\{e_1, \ldots, e_{k-1}, e\} \in \mathcal{U}$, dann hätte die Greedy-Methode $e$ statt $e_k$ als nächstes Element gewählt. $\{e_1, \ldots e_{k-1}\}$ maximal unabhängig in $F'$ ist ein Widerspruch zu $(c)$, da $\{e'_1, \ldots, e'_k\}$ unabhängige Teilmenge von $F'$ größerer Kardinalität ist. □

Folgerung: Falls $(M,\mathcal{U})$ ein Matroid ist, dann haben alle Basen des Matroids die gleiche Kardinalität.

**Satz 2.17.** *Sei $(M,\mathcal{U})$ ein Unabhängigkeitssystem und $\mathcal{B}$ das Basissystem von $(M,\mathcal{U})$. Dann sind äquivalent:*

- $(a)$ *Die Greedy-Methode für Basissysteme liefert eine Optimallösung für das Problem $\Pi = \min\{w(B) : B \in \mathcal{B}\}$ für beliebige Gewichtsfunktion $w$ über $M$.*
- $(b)$ *$(M,\mathcal{U})$ ist ein Matroid.*

*Beweis.* Wir zeigen, dass die Greedy-Methode für Basissysteme genau dann eine Optimallösung für das Problem $\min\{w(B) : B \in \mathcal{B}\}$ bei beliebiger Gewichtsfunktion $w$ liefert, wenn die Greedy-Methode für Unabhängigkeitssysteme für das Problem $\max\{w(I) : I \in \mathcal{U}\}$ bei beliebiger Gewichtsfunktion eine Optimallösung liefert.

$\Rightarrow$ Betrachte das Optimierungsproblem über dem Unabhängigkeitssystem $(M = \{e_1, \ldots, e_n\},\mathcal{U})$ mit $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_n)$ . Die Greedy-Methode für Basissysteme liefere für das

Basissystem $(M, \mathcal{B})$ bei beliebiger Gewichtsfunktion eine Optimallösung, also auch für das Problem $\min\{w'(B) : B \in \mathcal{B}\}$ mit

$$w'(e) := \begin{cases} -w(e) & \text{für } w(e) > 0 \\ 0 & \text{sonst} \end{cases}$$

Dann ist $w'(e_1) \leq w'(e_2) \leq \ldots \leq w'(e_n)$ und die Greedy-Methode liefert eine Basis $B^* \in \mathcal{B}$ mit $w'(B^*) \leq w'(B)$ für alle $B \in \mathcal{B}$. Die Greedy-Methode für Unabhängigkeitssysteme liefert dann als Lösung die Menge $I := \{e \in B^* : \ w'(e) < 0\}$. Wenn nun eine Menge $I' \in \mathcal{U}$ existiert mit $w(I') > w(I)$ und $w(I') \geq w(J)$ für alle $J \in \mathcal{U}$, dann existiert eine Basis $B' \in \mathcal{B}$ mit $I' \subseteq B'$ und $w(B') \leq w(I')$. D.h. $B' \setminus I' = \{e \in B' : \ w(e) \leq 0\}$. Also ist $w'(B') = w'(I') = -w(I') < -w(I) = w'(I) = w'(B^*)$. Widerspruch.

$\Leftarrow$ Betrachte nun das Optimierungsproblem über dem Basissystem $(M = \{e_1, \ldots, e_n\}, \mathcal{U})$ mit $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$. Die Greedy-Methode für Unabhängigkeitssystem liefere für das zugehörige Unabhängigkeitssystem $(M, \mathcal{U})$ bei beliebiger Gewichtsfunktion eine Optimallösung, also auch für das Problem $\max\{w'(I) : \ I \in \mathcal{U}\}$ mit: $w'(e) = m - w(e)$, wobei $m := \max_{e \in M}\{w(e) + 1\}$. Dann ist $w'(e_1) \geq w'(e_2) \geq \ldots \geq w'(e_n) > 0$, und die Greedy-Methode liefert die Menge $I^* \in \mathcal{U}$ mit $w'(I^*) \geq w'(I)$ für alle $I \in \mathcal{U}$. Darüberhinaus ist $I^* \in \mathcal{B}$, da $w'(e) > 0$ für alle $e \in M$. Die Greedy-Methode für Basissysteme liefert für das Problem $\min\{w(B) : \ B \in \mathcal{B}\}$ ebenfalls $I^*$. Es gilt nun

$$w(I^*) = \sum_{e_i \in I^*} (m - w'(e_i)) = |I^*|m - w'(I^*) \leq |I^*|m - w'(I) \text{ für alle } I \in \mathcal{U}$$

also auch für alle $B \in \mathcal{B}$ gilt

$$w(I^*) \leq |I^*|m - w'(B)$$

Da $|B| = |B'|$ für alle $B, B' \in \mathcal{B}$ und $I^* \in \mathcal{B}$ folgt

$$w(I^*) \leq |B| \cdot m - w'(B) = w(B) \text{ für alle } B \in \mathcal{B}$$

und damit die Behauptung. □

Bemerkung: Die Greedy-Methode (für Basis- bzw. Unabhängigkeitssysteme) ist optimal bezüglich beliebigen Gewichtsfunktionen genau dann, wenn sie optimal bezüglich 0/1-Gewichtsfunktionen ist.

# Kapitel 3

# Schnitte in Graphen und Zusammenhang

**Der Literaturtip.** Schnitte in Graphen und Zusammenhang werden in [19, 15, 8] beschrieben.

## 3.1 Schnitte minimalen Gewichts: MinCut

**Problem (MinCut).** *Gegeben sei ein Graph $G = (V, E)$ mit einer Kantengewichtsfunktion $c : E \longrightarrow \mathbb{R}_0^+$. Finde einen* nichttrivialen Schnitt $(S, V \backslash S)$ minimalen Gewichts in $G$, *d.h. finde $S \subseteq V, \emptyset \neq S \neq V$, so dass*

$$c(S, V \setminus S) := \sum_{\substack{\{u,v\} \in E, \\ u \in S, \\ v \in V \setminus S}} c(\{u, v\})$$

*minimal wird.* $(S, V \backslash S)$ *wird* minimaler Schnitt *genannt.*

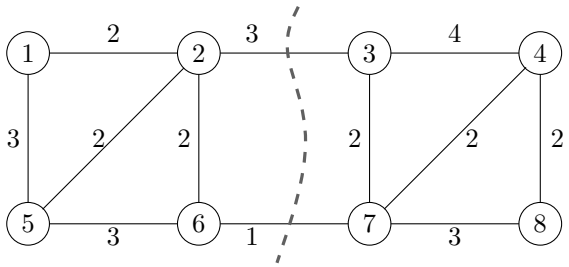



Abbildung 3.1: Beispiel eines minimalen Schnitts

**Bemerkung 3.1.** *Mit einem Flussalgorithmus (Ford & Fulkerson, Goldberg & Tarjan) kann man zu gegebenen $s$ und $t$ einen minimalen $s$–$t$–Schnitt bestimmen. Einen minimalen Schnitt allgemein kann man also durch $|V|^2$ Durchläufe eines Flussalgorithmus (für alle möglichen $s, t \in V$) berechnen oder sogar effizienter durch $|V| - 1$ Durchläufe, indem man ein $s$ festhält. In diesem Kapitel wollen wir einen minimalen Schnitt ohne Anwendung von Flussmethoden berechnen. Der*

*hier behandelte Algorithmus ist gleichzeitig (etwas) effizienter als $|V|-1$–maliges Anwenden des effizientesten bekannten Flussalgorithmus.*

Wir benutzen folgende Definitionen:

**Definition 3.2.** *Zu $S \subseteq V$ und $v \notin S$ sei*

$$c(S,v) := \sum_{\substack{\{u,v\} \in E \\ u \in S}} c(\{u,v\}).$$

*Den Knoten $v \in V \setminus S$, für den $c(S,v)$ maximal wird, nennen wir auch den* am stärksten mit $\boldsymbol{S}$ verbundenen Knoten.

**Definition 3.3 (Verschmelzen zweier Knoten).** *Seien $s,t \in V$. Dann werden $s$ und $t$* verschmolzen, *indem ein neuer Knoten $x_{s,t}$ eingeführt wird, $s$ und $t$ gelöscht werden, alle Nachbarn von $s$ und $t$ zu Nachbarn von $x_{s,t}$ werden und gegebenenfalls Kantengewichte von Kanten, die inzident zu $s$ oder $t$ waren, addiert werden. Falls zuvor $\{s,t\}$ eine Kante war, wird diese ebenfalls gelöscht.*

**Formal:** Sei $G=(V,E)$, $c: E \longrightarrow \mathbb{R}_0^+$, $s,t \in V$, $s \neq t$. Durch Verschmelzen von $s$ und $t$ wird $G$ in $G'=(V',E')$ und die Kantengewichtsfunktion in $c: E' \longrightarrow \mathbb{R}_0^+$ transformiert mit

$$\begin{aligned}
V' &:= \big(V \setminus \{s,t\}\big) \cup \{x_{s,t}\} \text{ mit } x_{s,t} \notin V \\
E' &:= E \setminus \{\{u,v\} \in E : \text{ wobei entweder } u=s \text{ oder } u=t\} \cup \\
&\quad \{\{x_{s,t},v\} : \{s,v\} \in E \text{ oder } \{t,v\} \in E \text{ und jeweils } v \in V \setminus \{s,t\}\} \\
c'(\{x_{s,t},v\}) &:= \begin{cases} c(\{s,v\}), & \text{falls } \{s,v\} \in E \text{ und } \{t,v\} \notin E \\ c(\{t,v\}), & \text{falls } \{t,v\} \in E \text{ und } \{s,v\} \notin E \\ c(\{s,v\}) + c(\{t,v\}), & \text{falls } \{s,v\},\{t,v\} \in E. \end{cases}
\end{aligned}$$

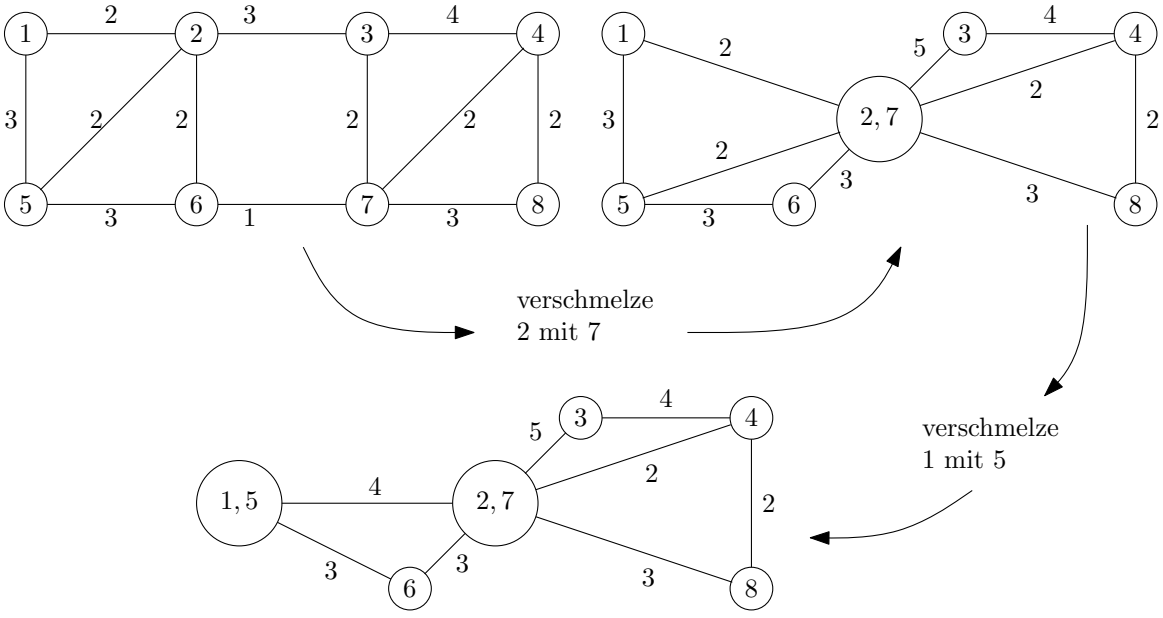



Abbildung 3.2: Ein Beispiel für das Verschmelzen von Knoten

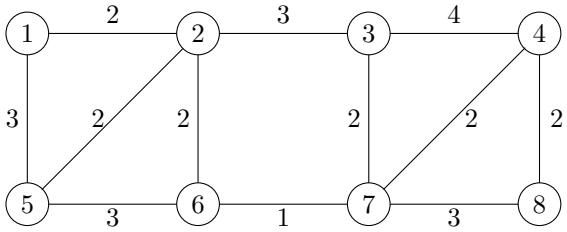
## 3.2 Der Algorithmus von Stoer & Wagner

Der folgende Algorithmus wurde von Stoer & Wagner (1994) veröffentlicht und basiert teilweise auf Ideen von Nagamochi & Ibaraki (1992).

Der Algorithmus besteht aus $|V| - 1$ Phasen. In der $i$–ten Phase wird in einem Graph $G_i$ ein Schnitt berechnet – der *Schnitt der Phase i*. Dieser Graph $G_i$ entsteht aus dem Graphen $G_{i-1}$ der der vorherigen Phase, durch Verschmelzen „geeigneter Knoten“ $s$ und $t$. Der Schnitt der Phase $i$ $(S_i, V_i \setminus S_i)$ wird mit einer Prozedur berechnet, die dem Algorithmus von Prim für MST entspricht. Ausgehend von einem Startknoten $a$ wird in jedem Schritt der am stärksten mit $S_i$ verbundene Knoten zu $S_i$ hinzugefügt, wobei zu Beginn $S_i := \{a\}$ ist. Die zu verschmelzenden „geeigneten Knoten“ $s$ und $t$ der Phase $i$ sind die beiden letzten Knoten, die zu $S_i$ hinzugefügt werden. Schnitt der Phase $i$ ist $S_i := (V \setminus \{t\}, \{t\})$. Ergebnis des ganzen Algorithmus ist der minimale Schnitt aller Schnitte der einzelnen Phasen $i$ $(1 \leq i \leq |V| - 1)$.

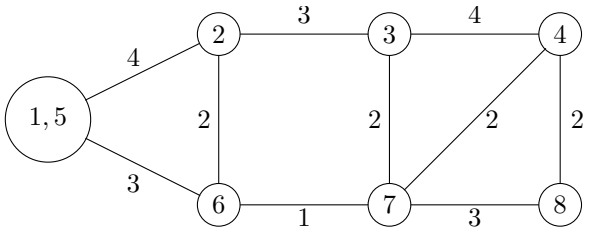
**Beispiel 3.4.** Der Startknoten sei 2.

- 1. Phase




  - $G_1 := G$.
  - $S_1 := \{2\}$
  - $S_1 = \{2, 3\}$.
  - $S_1 = \{2, 3, 4\}$
  - $S_1 = \{2, 3, 4, 7\}$
  - $S_1 = \{2, 3, 4, 7, 8\}$
  - $S_1 = \{2, 3, 4, 7, 8, 6\}$
  - $S_1 = \{2, 3, 4, 7, 8, 6, 5\} \Longrightarrow s = 5$
  - $S_1 = V_1$, also $t = 1$
  - Schnitt der ersten Phase ist also $(V_1 \setminus \{1\}, \{1\})$ mit Gewicht 5.

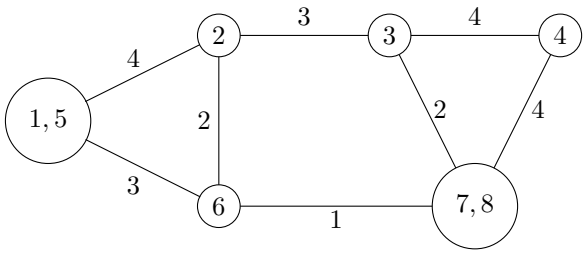
- 2. Phase




  - $S_2 := \{2\}$
  - $S_2 = \{2, \{1, 5\}\}$
  - $S_2 = \{2, \{1, 5\}, 6\}$

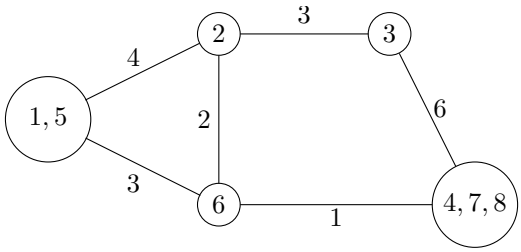
- $S_2 = \{2, \{1,5\}, 6, 3\}$
- $S_2 = \{2, \{1,5\}, 6, 3, 4\}$
- $S_2 = \{2, \{1,5\}, 6, 3, 4, 7\} \Longrightarrow s = 7$
- $S_2 = V_2$, also $t = 8$
- Schnitt der zweiten Phase ist also $(V_2 \setminus \{8\}, \{8\})$ mit Gewicht 5.

- 3. Phase



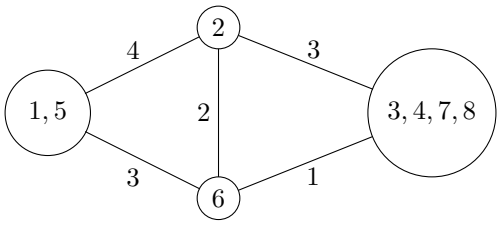

- $S_3 := \{2\}$
- $S_3 = \{2, \{1,5\}\}$
- $S_3 = \{2, \{1,5\}, 6\}$
- $S_3 = \{2, \{1,5\}, 6, 3\}$
- $S_3 = \{2, \{1,5\}, 6, 3, 4\} \Longrightarrow s = 4$
- $S_3 = V_3, t = \{7,8\}$
- Schnitt der dritten Phase ist also $(V_3 \setminus \{7,8\}, \{7,8\})$ mit Gewicht 7.

- 4. Phase



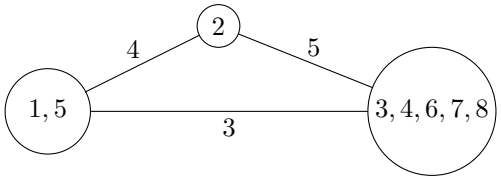

- $S_4 := \{2\}$
- $S_4 = \{2, \{1,5\}\}$
- $S_4 = \{2, \{1,5\}, 6\}$
- $S_4 = \{2, \{1,5\}, 6, 3\} \Longrightarrow s = 3$
- $S_4 = V_4, t = \{4,7,8\}$
- Schnitt der vierten Phase ist also $(V_4 \setminus \{4,7,8\}, \{4,7,8\})$ mit Gewicht 7.

- 5. Phase



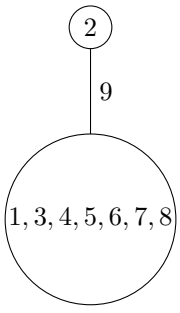
- $S_5 := \{2\}$
- $S_5 = \{2, \{1, 5\}\}$
- $S_5 = \{2, \{1, 5\}, 6\} \Longrightarrow s = 6$
- $S_5 = V_5, t = \{3, 4, 7, 8\}$
- Schnitt der fünften Phase ist also $(V_5 \setminus \{3, 4, 7, 8\}, \{3, 4, 7, 8\})$ mit Gewicht 4.

- 6. Phase




- $S_6 := \{2\}$
- $S_6 = \{2, \{3, 4, 6, 7, 8\}\} \Longrightarrow s = \{3, 4, 6, 7, 8\}$
- $S_6 = V_6, t = \{1, 5\}$
- Schnitt der sechsten Phase ist also $(V_6 \setminus \{1, 5\}, \{1, 5\})$ mit Gewicht 7.

- 7. Phase




- $S_7 := \{2\} \Longrightarrow S = 2$
- $S_7 = V_7, t = V \setminus \{2\}$
- Schnitt der siebten Phase ist also $(V_7 \setminus \{2\}, \{2\})$ mit Gewicht 9.

Der minimale Schnitt unter allen Schnitten der Phasen ist der Schnitt der fünften Phase mit Gewicht 4. Dieser Schnitt $(V_5 \setminus \{3, 4, 7, 8\}, \{3, 4, 7, 8\})$ in $G_5$ *induziert in* $G$ den Schnitt $(\{1, 2, 5, 6\}, \{3, 4, 7, 8\})$. ■

**Formale Beschreibung des Algorithmus:**

**Algorithmus 31** : MINSCHNITTPHASE$(G_i, c, a)$

```
1 S ← {a}
2 t ← a
3 Solange S ≠ V_i tue
4 |   Bestimme Knoten v ∈ V_i \ S mit c(S, v) maximal und S ← S ∪ {v}
5 |   s ← t
6 |   t ← v
7 Speichere (V_i \ {t}, {t}) als SCHNITT–DER–PHASE
8 Konstruiere aus G_i Graph G_{i+1} durch Verschmelzen von s und t
```

**Algorithmus 32** : MIN–SCHNITT$(G, c, a)$

```
1 G_1 ← G
2 Für i = 1 bis |V| − 1
3 |   MINSCHNITTPHASE(G_i, c, a)
4 |   Wenn SCHNITT–DER–PHASE ist kleiner als MIN–SCHNITT (der bisher minimale
  |   SCHNITT–DER–PHASE)
5 |   |   speichere SCHNITT–DER–PHASE als MIN–SCHNITT
6 Gib MIN–SCHNITT aus
```

**Laufzeit:** Die Prozedur MINSCHNITTPHASE kann genauso wie der Algorithmus von Prim implementiert werden, wobei nur anstatt eines Minimums jeweils ein Maximum berechnet und am Ende $G_{i+1}$ konstruiert werden muss. Mit einem HEAP kann Prozedur MINSCHNITTPHASE in $\mathcal{O}((|E|+|V|)\log|V|)$ ausgeführt werden. Dazu werden die Knoten außerhalb der Menge $S$ in einem HEAP verwaltet, in dessen Wurzel das Element mit maximalem Schlüsselwert steht. Der Schlüsselwert von $v \in V \setminus S$ ist jeweils $c(S, v)$ zur aktuellen Menge $S$. Wird ein Knoten zu $S$ hinzugefügt, so wird die Wurzel des HEAP gelöscht. Zudem werden die Schlüsselwerte der Knoten im HEAP, welche im Graphen adjazent zum gelöschten Knoten sind, jeweils entsprechend erhöht und die Knoten dann an die richtige Stelle im HEAP bewegt. Ein Aufruf von MINSCHNITTPHASE erfordert also maximal $|V|$-mal DELETEMAX und höchstens $|E|$-mal INCREASE-KEY mit den entsprechenden Knotenbewegungen im HEAP. Diese Knotenbewegungen im HEAP sind jeweils in $O(\log|V|)$. Verwendet man einen sog. FIBONACCI-HEAP, ist der amortisierte Aufwand von INCREASE-KEY sogar in $O(1)$. Damit ist der Aufwand für MINSCHNITTPHASE in $\mathcal{O}(|V|\log|V| + |E|)$ und der Aufwand von MINSCHNITT insgesamt in $\mathcal{O}(|V|^2\log|V| + |V||E|)$.

**Korrektheit des Algorithmus:** Für $s, t \in V$, $s \neq t$ nenne den Schnitt $(S, V \setminus S)$ mit $s \in S$ und $t \in V \setminus S$ einen ***s*–*t*–*Schnitt***. Ein *s*–*t*–Schnitt *trennt* Knoten $u$ und $v$, wenn $u \in S$ und $v \in V \setminus S$.

**Lemma 3.5.** *Zu $G = (V, E)$ und $c : E \longrightarrow \mathbb{R}_0^+$ gilt für die Prozedur* MINSCHNITTPHASE$(G, c, a)$ *mit beliebigem $a \in V$, dass der berechnete* SCHNITT-DER-PHASE *minimal ist unter allen $s$–$t$–Schnitten, wobei $s$ und $t$ vorletzter bzw. letzter betrachteter Knoten ist.*

*Beweis.* Sei $(S, V \setminus S)$ SCHNITT-DER-PHASE, $s$ und $t$ vorletzter und letzter betrachteter Knoten. MINSCHNITTPHASE betrachtet die Knoten aus $V$ entsprechend einer „linearen Ordnung“, die mit $a$ beginnt und mit $s$ und $t$ endet. Betrachte einen beliebigen $s$-$t$-Schnitt $(S', V \setminus S')$. Wir zeigen, daß $c(S', V \setminus S') \geq c(S, V \setminus S)$ ist. Nenne einen Knoten $v \in V$ *aktiv* (bzgl. $S'$), wenn $v$ und der Knoten, der unmittelbar vor $v$ von MINSCHNITTPHASE zu $S$ hinzugefügt wurde, durch den Schnitt $(S', V \setminus S')$ getrennt werden. Zu $v \in V \setminus \{a\}$ sei $S_v$ die Menge aller Knoten, die vor $v$ zu $S$ hinzugefügt wurden und $S'_v := S' \cap (S_v \cup \{v\})$.

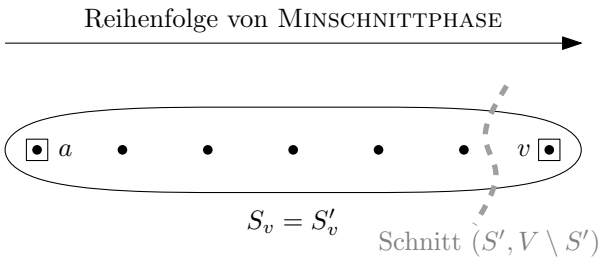
Betrachte nun den durch $S'$ induzierten Schnitt in dem durch $S_v \cup \{v\}$ induzierten Graphen. Wir zeigen, dass für alle aktiven Knoten $v$ gilt:

$$\begin{aligned} c(S_v, v) &\leq c(S' \cap (S_v \cup \{v\}), S_v \cup \{v\} \setminus (S' \cap (S_v \cup \{v\})) \\ &= c(S'_v, (S_v \cup \{v\}) \setminus S') \\ &= c(S'_v, (S_v \cup \{v\}) \cap (V \setminus S')) \end{aligned}$$

Der Beweis wird durch Induktion über die aktiven Knoten $v$ in der Reihenfolge, in der sie zu $S$ hinzugefügt werden, geführt.

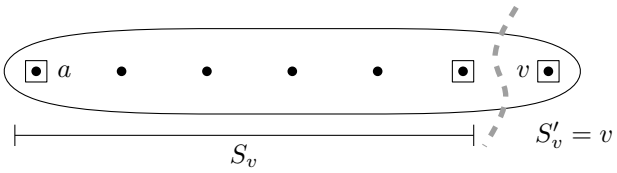
**Induktionsanfang:** Sei $v$ erster aktiver Knoten, wir unterscheiden ob $v$ in $S'$ ist, oder nicht:

- $v \notin S'$:




  Da $v$ der erste aktive Knoten ist und $v \notin S'$ gilt $S'_v = S_v$ und weiterhin $(V \setminus S') \cap (S_v \cup \{v\}) = \{v\}$.

- $v \in S'$:

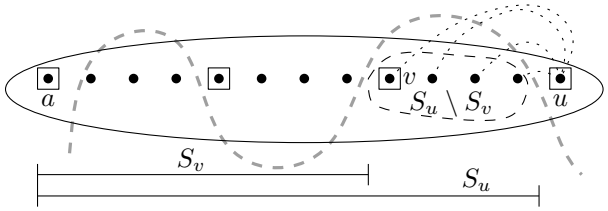



  Der Knoten $v$ ist nun bei MinSchnittphase der erste Knoten der in $S'$ liegt, somit gilt $S'_v = \{v\}$ und $(V \setminus S') \cap (S_v \cup \{v\}) = S_v$.

In beiden Fällen gilt die Behauptung:

$$c(S_v, v) = c(S'_v, V \setminus S' \cap (S_v \cup \{v\}).$$

**Induktionsschritt:** Angenommen, die Behauptung gelte für alle aktiven Knoten bis zum Knoten $v$ und $u$ sei der nächste aktive Knoten.




Dann gilt zunächst offenbar:

$$c(S_u, u) = c(S_v, u) + c(S_u \setminus S_v, u). \tag{3.1}$$

Da $v$ vor $u$ zu $S$ hinzugefügt wurde, gilt, da MINSCHNITTPHASE stets den am stärksten verbundenen Knoten hinzufügt

$$c(S_v, u) \leq c(S_v, v). \tag{3.2}$$

Außerdem ist nach Induktionsannahme

$$c(S_v, v) \leq c(S'_v, V \setminus S' \cap (S_v \cup \{v\})). \tag{3.3}$$

Setzt man nun Ungleichung 3.3 und Ungleichung 3.2 in Gleichung 3.1 ein, so erhält man

$$\begin{aligned} c(S_u, u) &\leq c(S_v, v) + c(S_u \setminus S_v, u) \\ &\leq c(S'_v, (V \setminus S') \cap (S_v \cup \{v\})) + c(S_u \setminus S_v, u) \end{aligned}$$

Wegen der Definition aktiver Knoten liegen die aufeinanderfolgenden aktiven Knoten $u$ und $v$ in unterschiedlichen Seiten des Schnitts $(S', V \setminus S')$. Somit verbinden alle Kanten zwischen Knoten in $S_u \setminus S_v$ und dem Knoten $u$ die verschiedenen Seiten von $S'$. Daher tragen sie zu $c(S'_u, (V \setminus S') \cap (S_u \cup \{u\}))$ bei, dem durch $S'$ induzierten Schnitt in dem durch $S_u \cup \{u\}$ induzierten Graphen. Ebenso tragen zu diesem Schnitt aber auch all die Kanten bei, welche von dem Schnitt gekreuzt werden, der durch $S'$ in dem durch $S_v \cup \{v\}$ induzierten Graphen induziert wird, da $(S_v \cup \{v\}) \subset (S_u \cup \{u\})$ (die Grundmenge auf der $S'$ schneidet ist echt größer geworden). Identifizieren wir einen Schnitt mit der Menge von Kanten, welche diesen Schnitt kreuzen, so können wir also formal festhalten:

$$\begin{aligned} (S_u \setminus S_v, u) &\subseteq (S'_u, (V \setminus S') \cap (S_u \cup \{u\})) && \text{und} \\ (S'_v, (V \setminus S') \cap (S_v \cup \{v\})) &\subseteq (S'_u, (V \setminus S') \cap (S_u \cup \{u\})) && \text{zudem gilt noch} \\ (S_u \setminus S_v, u) \cap (S'_v, (V \setminus S') \cap (S_v \cup \{v\})) &= \emptyset && \text{(Disjunktheit).} \end{aligned}$$

Somit gilt also zusammenfassend

$$\begin{aligned} c(S_u, u) &\leq c(S'_v, (V \setminus S') \cap (S_v \cup \{v\})) + c(S_u \setminus S_v, u) \\ &\leq c(S'_u, (V \setminus S') \cap (S_u \cup \{u\})). \end{aligned}$$

Durch vollständige Induktion wurde nun gezeigt, dass für alle aktiven Knoten $v$ gilt $c(S_v, v) \leq c(S'_v, (S_v \cup \{v\}) \cap (V \setminus S')$. Da $(S', V \setminus S')$ein $s$-$t$-Schnitt ist, ist $t$, der letzte Knoten in der linearen Ordnung von MINSCHNITTPHASE, ein aktiver Knoten bezüglich $S'$. Es folgt also

$$c(S, V \setminus S) \underbrace{=}_{\text{MINSCHNITTPHASE}} c(S_t, t) \underbrace{\leq}_{\text{V.I.}} c(S'_t, (V \setminus S') \cap (S_t \cup \{t\})) \underbrace{=}_{S_t \cup \{t\} = V} c(S', V \setminus S'). \qquad \square$$

**Satz 3.6.** *Der minimale Schnitt von allen Ergebnissen der $|V| - 1$ Ausführungen von* MINSCHNITTPHASE *ist ein minimaler, nichttrivialer Schnitt in $G = (V, E)$ mit $|V| \geq 2$.*

*Beweis.* Induktion über $|V|$.
**Induktionsanfang:** $|V| = 2$ ist trivial.
**Induktionsschritt:** Sei $|V| \geq 3$. Betrachte Phase 1: Falls $G$ einen nichttrivialen minimalen Schnitt hat, der gerade $s$ und $t$ (vorletzter bzw. letzter Knoten der Phase 1) trennt, so ist der SCHNITT-DER-PHASE 1 nach Lemma 3.5 ein minimaler nichttrivialer Schnitt. Wenn es jedoch keinen nichttrivialen minimalen Schnitt in $G$ gibt, der $s$ und $t$ trennt, so müssen in jedem minimalen Schnitt $s$ und $t$ in derselben Menge liegen. Der Graph $G'$, der aus $G$ durch Verschmelzen von $s$ und $t$ entsteht, hat also einen Schnitt, der gleichzeitig einen minimalen Schnitt in $G$ induziert. Es genügt also, einen minimalen Schnitt von $G'$ zu bestimmen. Diesen bestimmt der Algorithmus nach Induktionsannahme mit dem Durchlaufen der Phasen 2 bis $|V|-1$, da $G' = (V', E')$ die Ungleichung $|V'| < |V|$ erfüllt. $\square$

**Bemerkung 3.7.** *Bei allgemeinerer Kantengewichtsfunktion $c : E \longrightarrow \mathbb{R}$ (d.h. negative Gewichte sind zugelassen) ist das* MINCUT*-Problem im Allgemeinen* NP*-schwer. Ebenso ist das „duale“* MAXCUT*-Problem im allgemeinen* NP*-schwer.*

Das ungewichtete MinCut-Problem ist äquivalent zu folgendem grundlegenden Graphenproblem:

> Was ist die minimale Anzahl an Kanten in $G = (V, E)$, deren Wegnahme einen unzusammenhängenden Graphen induziert?

Der entsprechende Wert wird auch *Kanten-Zusammenhangs-Zahl* genannt.

# Kapitel 4

# Flussprobleme und Dualität

## 4.1 Grundlagen

**Definition 4.1.** *Sei ein einfacher gerichteter Graph $D = (V, E)$ mit* Kantenkapazitäten $c : E \to \mathbb{R}_0^+$ *und ausgezeichneten Knoten $s, t \in V$, $s$* Quelle (source) *und $t$* Senke (target) *gegeben. Man bezeichnet das Tupel $(D; s, t; c)$ dann als* Netzwerk. *Eine Abbildung $f : E \to \mathbb{R}_0^+$ heißt* Fluss, *wenn sie die folgenden beiden Eigenschaften hat:*

1. *Für alle $(i, j) \in E$ ist die* Kapazitätsbedingung
$$0 \leq f(i, j) \leq c(i, j) \tag{4.1}$$
*erfüllt.*
2. *Für alle $i \in V \setminus \{s, t\}$ ist die* Flusserhaltungsbedingung
$$\sum_{\{j|(i,j)\in E\}} f(i, j) - \sum_{\{j|(j,i)\in E\}} f(j, i) = 0 \tag{4.2}$$
*erfüllt.*

**Lemma 4.2.** *Für einen Fluss $f$ in einem Netzwerk $(D; s, t; c)$ gilt*
$$\sum_{(s,i)\in E} f(s, i) - \sum_{(i,s)\in E} f(i, s) = \sum_{(i,t)\in E} f(i, t) - \sum_{(t,i)\in E} f(t, i) \ .$$

*Beweis.* Es gilt
$$\begin{aligned}\sum_{(i,j)\in E} f(i, j) &= \sum_{(i,s)\in E} f(i, s) + \sum_{(i,t)\in E} f(i, t) + \sum_{j\in V\setminus\{s,t\}} \sum_{(i,j)\in E} f(i, j) \\ &= \sum_{(s,i)\in E} f(s, i) + \sum_{(t,i)\in E} f(t, i) + \sum_{j\in V\setminus\{s,t\}} \sum_{(j,i)\in E} f(j, i) \ .\end{aligned}$$
Wegen der Flusserhaltungsbedingung (4.2) sind die letzten Terme der beiden Zeilen gleich und somit folgt die Behauptung. □

**Definition 4.3.** *Der Ausdruck*
$$w(f) := \sum_{(s,i)\in E} f(s, i) - \sum_{(i,s)\in E} f(i, s)$$
*heißt* Wert *des Flusses $f$.*

*Ein Fluss $f$, für den $w(f)$ maximal ist, d.h. $w(f') \leq w(f)$ für alle Flüsse $f'$ in Netzwerk $(D; s, t; c)$, heißt* Maximalfluss *in $(D; s, t; c)$.*

### 4.1.1 Problemstellung

Die grundlegende Problemstellung besteht darin, in einem Netzwerk $(D; s, t; c)$ einen Maximalfluss zu finden.

**Definition 4.4.** *Eine Menge $S \subset V$ induziert eine Partition $(S, V \setminus S)$ der Knotenmenge $V$, die wir* Schnitt *im Graphen $D = (V, E)$ nennen. In einem Netzwerk $(D; s, t; c)$ heißt $(S, V \setminus S)$ ein $s$-$t$-Schnitt, wenn $s \in S$ und $t \in V \setminus S$. Die* Kapazität *eines Schnittes $(S, V \setminus S)$ ist definiert als*

$$c(S, V \setminus S) := \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in V\setminus S}} c(i,j) \ .$$

*Ein Schnitt $(S, V \setminus S)$ heißt* minimal, *wenn $c(S, V \setminus S)$ minimalen Wert unter allen Schnitten $(S', V \setminus S')$ in $D$ hat, d.h. $c(S', V \setminus S') \geq c(S, V \setminus S)$ für alle $S' \subset V$ mit $\emptyset \neq S' \neq V$.*

**Lemma 4.5 (Schnitt-Lemma).** *Sei $(S, V \setminus S)$ ein $s$-$t$-Schnitt im Netzwerk $(D; s, t; c)$. Für jeden Fluss $f$ gilt, dass*

$$w(f) = \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in V\setminus S}} f(i,j) - \sum_{\substack{(i,j)\in E \\ j\in S \\ i\in V\setminus S}} f(i,j) \ .$$

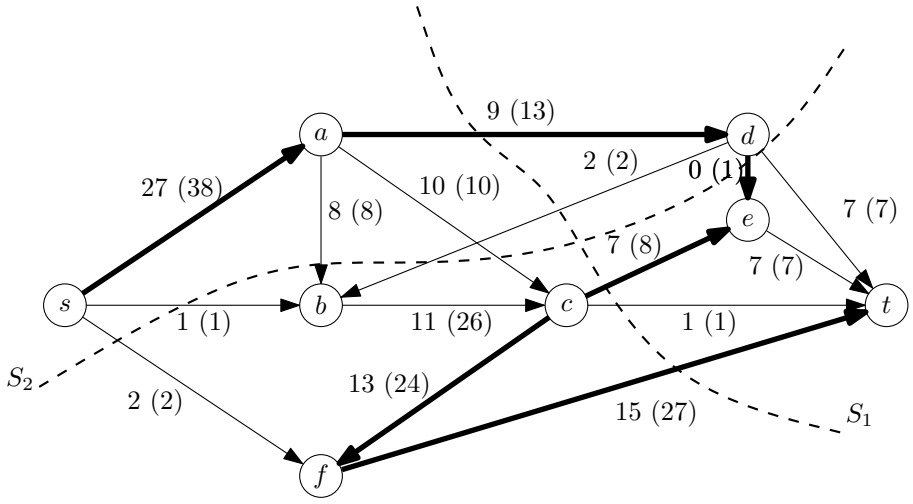
*Insbesondere ist $w(f) \leq c(S, V \setminus S)$.*

*Beweis.* Es gilt

$$\begin{aligned} w(f) &= \sum_{(s,i)\in E} f(s,i) - \sum_{(i,s)\in E} f(i,s) + \underbrace{\sum_{\substack{(i,j)\in E \\ i\in S\setminus\{s\}}} f(i,j) - \sum_{\substack{(j,i)\in E \\ i\in S\setminus\{s\}}} f(j,i)}_{=0} \\ &= \sum_{i\in S}\Big( \sum_{(i,j)\in E} f(i,j) - \sum_{(j,i)\in E} f(j,i)\Big) \\ &= \underbrace{\sum_{\substack{(i,j)\in E \\ i,j\in S}} f(i,j) - \sum_{\substack{(j,i)\in E \\ i,j\in S}} f(j,i)}_{=0} + \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in V\setminus S}} f(i,j) - \underbrace{\sum_{\substack{(j,i)\in E \\ i\in S \\ j\in V\setminus S}} f(j,i)}_{\geq 0} \\ &\leq \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in V\setminus S}} c(i,j) = c(S, V \setminus S) \end{aligned}$$

□

**Beispiel 4.6.** Die folgende Abbildung zeigt ein Flussnetzwerk mit einem eingezeichneten Fluss mit Wert 30. Die Kapazitäten der Kanten sind in Klammern angegeben. Außerdem sind zwei $s$-$t$-Schnitte $S_1$ und $S_2$ mit den Kapazitäten 49 und 31 eingezeichnet, wobei letzterer minimal ist. Der Schlüssel zum Finden eines Flusses mit größerem Wert scheint ein erhöhender Weg von $s$ nach $t$ zu sein, bestehend aus Vorwärtskanten $(i, j)$, auf denen $f(i, j) < c(i, j)$ und Rückwärtskanten $(i, j)$, auf denen $f(i, j) > 0$. Die hervorgehobenen Kanten zeigen einen solchen erhöhenden Weg.




**Definition 4.7.** *Zu einem Fluss $f$ im Netzwerk $(D; s, t; c)$ betrachten wir einen (ungerichteten) Weg von $s$ nach $t$. Alle Kanten auf diesem Weg, die von $s$ in Richtung $t$ gerichtet sind, heißen* Vorwärtskanten*, alle anderen* Rückwärtskanten*. Ein solcher Weg heißt* erhöhender Weg *(bezüglich $f$), wenn für jede Vorwärtskante $(i, j)$ des Weges $f(i, j) < c(i, j)$ gilt und wenn für jede Rückwärtskante $f(i, j) > 0$.*

**Satz 4.8 (Satz vom erhöhenden Weg).** *Ein Fluss $f$ in einem Netzwerk $(D; s, t; c)$ ist genau dann ein Maximalfluss, wenn es bezüglich $f$ keinen erhöhenden Weg gibt.*

*Beweis.*

$\Longrightarrow$ : Sei $f$ ein Maximalfluss. Angenommen, es existiert bezüglich $f$ ein erhöhender Weg $W$. Sei für Kanten $(i, j)$ dieses Weges

$$\Delta(i,j) := \begin{cases} c(i,j) - f(i,j) & \text{falls } (i,j) \text{ Vorwärtskante} \\ f(i,j) & \text{falls } (i,j) \text{ Rückwärtskante} \end{cases}$$

und

$$\Delta := \min\{\Delta(i,j) \mid (i,j) \text{ auf erhöhendem Weg } W\} \ .$$

Dann ist $\Delta > 0$. Sei nun $f' : E \to \mathbb{R}_0^+$ definiert als

$$f' := \begin{cases} f(i,j) + \Delta & \text{falls } (i,j) \text{ Vorwärtskante auf } W \\ f(i,j) - \Delta & \text{falls } (i,j) \text{ Rückwärtskante auf } W \\ f(i,j) & \text{sonst .} \end{cases}$$

Dann ist $f'$ wieder ein Fluss und $w(f') > w(f)$ im Widerspruch zu der Annahme, dass $f$ ein Maximalfluss ist. □

$\Longleftarrow$ : Das Netzwerk $(D; s, t; c)$ habe keinen bezüglich $f$ erhöhenden Weg. Sei $S$ die Menge aller Knoten in $V$, zu denen ein erhöhender Weg von $s$ aus bezüglich $f$ existiert. Es gilt $S \neq \emptyset$, weil $s \in S$, und $S \neq V$, weil $t \notin S$. Dann induziert $S$ einen $s$-$t$-Schnitt und es muss gelten, dass $f(i, j) = c(i, j)$ für alle $(i, j)$ mit $i \in S$, $j \in V \setminus S$ und dass $f(i, j) = 0$ für alle $(i, j)$ mit $i \in V \setminus S$, $j \in S$ (d.h. alle Kanten $(i, j)$ mit $i \in S$, $j \in V \setminus S$ sind „saturiert“ und alle Kanten $(i, j)$ mit $i \in V \setminus S$, $j \in S$ sind „leer“). Nach Schnitt-Lemma 4.5 ergibt sich $w(f) = c(S, V \setminus S)$. Es muss also $w(f)$ maximal sein.

**Satz 4.9 („Max-Flow Min-Cut Theorem“ von Ford und Fulkerson 1956[1]).** *In einem Netzwerk $(D; s, t; c)$ ist der Wert eines Maximalflusses gleich der minimalen Kapazität eines $s$-$t$-Schnittes.*

*Beweis.* Die Behauptung folgt direkt aus dem Satz vom erhöhenden Weg 4.8. Denn ist $f$ ein Maximalfluss, dann existiert ein Schnitt $(S, V \setminus S)$ mit $s \in S$ und $t \in V \setminus S$ (wobei $S$ die Menge aller auf einem erhöhenden Weg von $s$ erreichbaren Knoten ist). Für $(S, V \setminus S)$ gilt

$$w(f) = c(S, V \setminus S) \quad \text{und} \quad c(S, V \setminus S') = \min_{\substack{s \in S' \\ t \in V \setminus S'}} c(S', V \setminus S') \ . \qquad \square$$

**Bemerkung 4.10.** *Für einen Fluss $f$ in einem Netzwerk $(D; s, t; c)$ sind folgende Aussagen äquivalent:*

1. *Der Wert $w(f)$ ist maximal.*
2. *Es gibt keinen bezüglich $f$ erhöhenden Weg.*
3. *Die Kapazität eines minimalen $s$-$t$-Schnittes $(S, V \setminus S)$ ist $w(f)$.*

**Satz 4.11 (Ganzzahligkeitssatz).** *Sei $(D; s, t; c)$ ein Netzwerk mit $c : E \to \mathbb{N}_0$. Dann gibt es einen Maximalfluss $f$ mit $f(i, j) \in \mathbb{N}_0$ für alle $(i, j) \in E$ und damit $w(f) \in \mathbb{N}_0$.*

*Beweis.* Wir definieren einen ganzzahligen „Anfangsfluss“ $f_0 : E \to \mathbb{N}_0$ (zum Beispiel $f_0(i, j) = 0$ für alle $(i, j) \in E$). Ist $f_0$ nicht maximal, so existiert ein erhöhender Weg bezüglich $f_0$, und für diesen ist $\Delta_0 > 0$ (definiert wie $\Delta$ in Satz 4.8) ganzzahlig. Entsprechend kann $f_1$ mit $w(f_0) = w(f_0) + \Delta_0$ konstruiert werden und $f_1$ ist wiederum ganzzahlig. Das Verfahren kann so lange iteriert werden, bis ein ganzzahliger Fluss $f_i$ erreicht ist, bezüglich dessen es keinen erhöhenden Weg mehr gibt. $\square$

## 4.2 Bestimmung maximaler Flüsse („Max Flow“)

Entsprechend dem Beweis des Satzes vom erhöhenden Weg 4.8 können wir folgendes Verfahren zur Bestimmung eines Maximalflusses (und eines minimalen Schnittes) angeben (dabei stehe VwK für Vorwärts- und RwK für Rückwärtskante): Einen erhöhenden Weg kann man systematisch mittels

**Algorithmus 33** : MAX-FLOW

**Eingabe** : Gerichteter Graph, Quelle, Senke, Kantenkapazitäten
**Ausgabe** : Maximalfluss von der Quelle zur Senke

```
1 f(i,j) ← 0 für alle Kanten (i,j) ∈ E
2 Solange Es gibt einen erhöhenden Weg bezüglich f tue
3     Sei ⟨e_1, e_2, ..., e_k⟩ mit e_1, ..., e_k ∈ E erhöhender Weg
4     Δ ← min({c(e_i) − f(e_i) | e_i VwK} ∪ {f(e_i) | e_i RwK})
5     ∀e_i ∈ E : f(e_i) ← f(e_i) + Δ, falls e_i eine Vorwärtskante ist
6     ∀e_i ∈ E : f(e_i) ← f(e_i) − Δ, falls e_i eine Rückwärtskante ist
7 Ende
```

einer Graphensuche finden.

**Algorithmus 34** : Algorithmus von Ford-Fulkerson

**Eingabe** : Ein Netzwerk $(D; s, t; c)$ mit dem Graph $D = (V, E)$, wobei $V := \{1, \ldots, n\}$, und Kapazitätsfunktion $c : E \to \mathbb{R}_0^+$

**Ausgabe** : Ein Maximalfluss $f$ und ein minimaler $s$-$t$-Schnitt $(S, V \setminus S)$

**Für** $(i, j) \in E$
  $f(i, j) \leftarrow 0$

Lege Datenstrukturen an:

- $S$ (Menge der markierten Knoten)
- VOR (Array der Länge $n - 1$, in dem für alle Knoten aus $V \setminus \{s\}$ der Vorgänger auf einem erhöhenden Weg von $s$ nach $t$ gespeichert wird)
- $\Delta$ (Array der Länge $n$ zur sukzessiven Bestimmung der „Erhöhungswerte“ $\Delta$)
- $u$ (Hilfsarray der Länge $n$ zur Durchführung der Suche)

$S \leftarrow \{s\}$
**Für** $v \in V$
  $u(v) \leftarrow$ False
  $\Delta(v) \leftarrow \infty$
**Solange** *Es gibt ein* $v \in S$ *mit* $u(v) =$ False **tue**
  Wähle $v \in S$ mit $u(v) =$ False
  **Für** $(v, w) \in E$ *mit* $w \notin S$
    **Wenn** $f(v, w) < c(v, w)$
      $\text{VOR}(w) \leftarrow +v$
      $\Delta(w) \leftarrow \min\{c(v, w) - f(v, w), \Delta(v)\}$
      $S \leftarrow S \cup \{w\}$
  **Für** $(w, v) \in E$ *mit* $w \notin S$
    **Wenn** $f(w, v) > 0$
      $\text{VOR}(w) \leftarrow -v$
      $\Delta(w) \leftarrow \min\{f(w, v), \Delta(v)\}.$
      $S \leftarrow S \cup \{w\}$
  $u(v) \leftarrow$ True
  **Wenn** $t \in S$
    $w \leftarrow t$
    **Solange** $w \neq s$ **tue**
      **Wenn** $\text{VOR}(w) > 0$
        $f(\text{VOR}(w), w) \leftarrow f(\text{VOR}(w), w) + \Delta(t)$
      **sonst**
        $f(w, -\text{VOR}(w)) \leftarrow f(w, -\text{VOR}(w)) - \Delta(t)$
      $w \leftarrow \text{VOR}(w)$
    $S \leftarrow \{s\}$
    **Für** $v \in V$
      $u(v) \leftarrow$ False
      $\Delta(v) \leftarrow \infty$

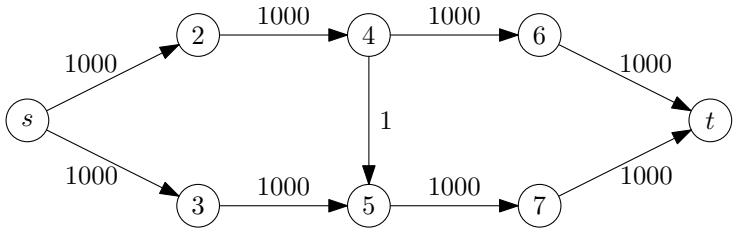
Gib $f$ und $(S, V \setminus S)$ aus

### 4.2.1 Ford-Fulkerson-Algorithmus (1962)

Der Algorithmus von Ford und Fulkerson (Algorithmus 34) berechnet einen maximalen Fluss mittels erhöhender Wege. In den Schritten 9 bis 20 wird ein erhöhender $s$-$t$-Weg gesucht. Dazu wird die Menge $S \subset V$ der Knoten bestimmt, die auf erhöhenden Wegen von $s$ aus erreichbar sind.

Laufzeit: Die Laufzeit des Algorithmus hängt davon ab, wie geschickt $v$ ausgewählt wird (in Schritt 9), und davon, wie oft erhöht wird. Die Anzahl der Erhöhungen hängt hier auch ab von $C := \max\{c(i,j) \mid (i,j) \in E\}$. Bei nichtrationalen Werten $c(i,j)$ kann es passieren, dass das Verfahren nicht terminiert. Bei rationalen Werten geht $C$ im Allgemeinen in die Laufzeit ein.

**Beispiel 4.12.**




Der Wert eines maximalen Flusses ist 2000. Es kann passieren, dass abwechselnd entlang der Wege $s \to 2 \to 4 \to 5 \to 7 \to t$ und $s \to 3 \to 5 \leftarrow 4 \to 6 \to t$ um jeweils eine Flusseinheit erhöht wird. ■

### 4.2.2 Der Algorithmus von Edmonds und Karp (1972)

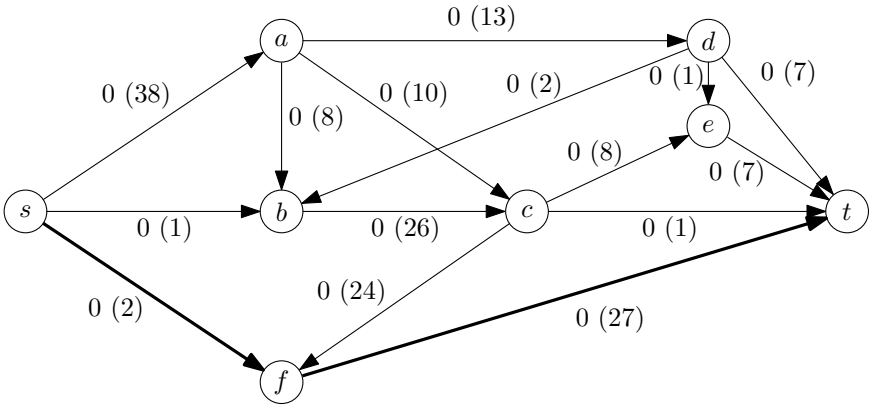
Um einen Fall wie in Beispiel 4.12 zu vermeiden, geht der Algorithmus von Edmonds und Karp geschickter vor. Dort wird der Schritt 9 des Algorithmus „wähle $v \in S$ mit $u(v) =$ False“ ersetzt durch “wähle unter allen $v \in S$ mit $u(v) =$ False das $v$ aus, welches schon am längsten in $S$ ist ($\rightsquigarrow$ Breitensuche).“

Dazu wird $S$ als QUEUE implementiert. Dieser Algorithmus kann dann in $O(|V||E|^2)$ implementiert werden. Der Fluss wird maximal $O(|V||E|)$ oft erhöht und die Erhöhung kostet jeweils höchstens $O(|E|)$.

#### Beispiel zum Algorithmus von Edmonds und Karp

Wir führen den Algorithmus von Edmonds und Karp an dem folgenden Beispielgraphen vor, der den Anfangsfluss (überall 0) und die Kapazitäten in Klammern zeigt. Hervorgehoben ist außerdem der erste erhöhende Weg, der gefunden werden könnte.

[1] ebenso Elias, Feinstein und Shannon 1956



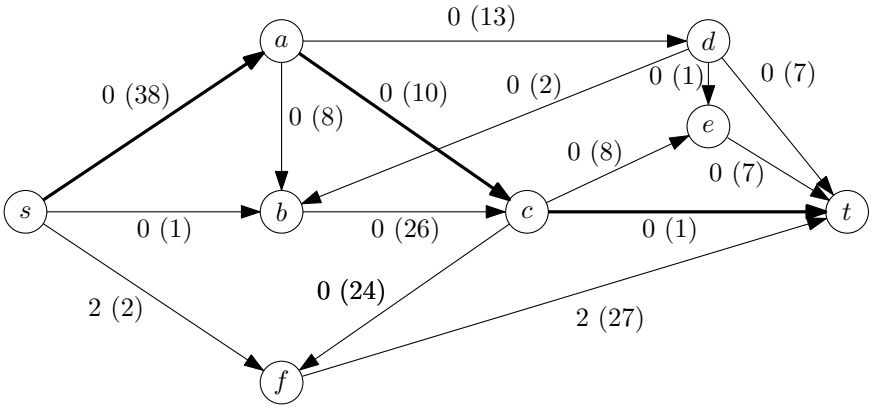

Der eingezeichnete erhöhende Weg wurde wie folgt gefunden: Zunächst ist $S = \{s\}$ und $u(v) =$ False für alle $v \in V$. Von $s$ aus werden die Knoten $a$, $b$ und $f$ erreicht, das heißt $S := \{s, a, b, f\}$ und

$$\begin{aligned} \mathrm{Vor}(a) &:= s & \Delta(a) &:= 38 \\ \mathrm{Vor}(b) &:= s & \Delta(b) &:= 1 \\ \mathrm{Vor}(f) &:= s & \Delta(f) &:= 2 \ . \end{aligned}$$

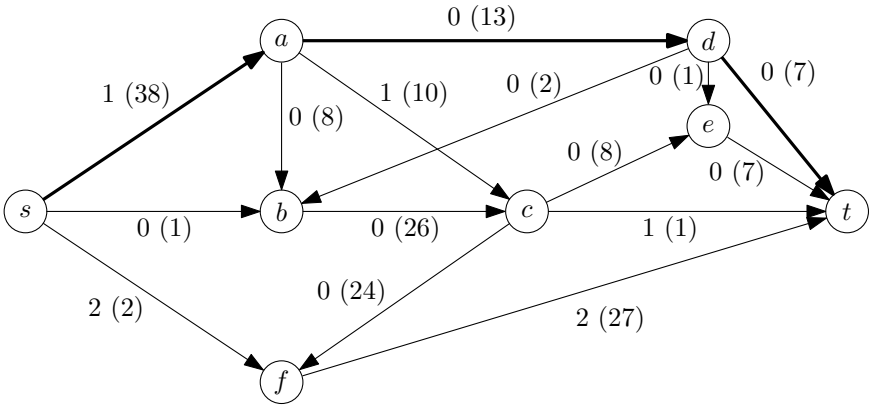
Nun werden von $a$ aus die Knoten $c$ und $d$ gefunden und von $f$ aus der Knoten $t$, womit $t \in S$ ist und

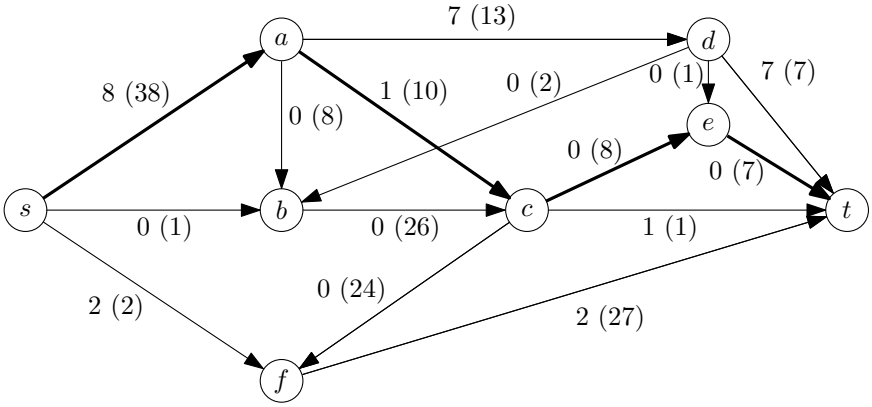
$$\begin{aligned} \mathrm{Vor}(c) &:= a & \Delta(c) &:= 10 \\ \mathrm{Vor}(d) &:= a & \Delta(d) &:= 13 \\ \mathrm{Vor}(t) &:= f & \Delta(t) &:= 2 \ . \end{aligned}$$

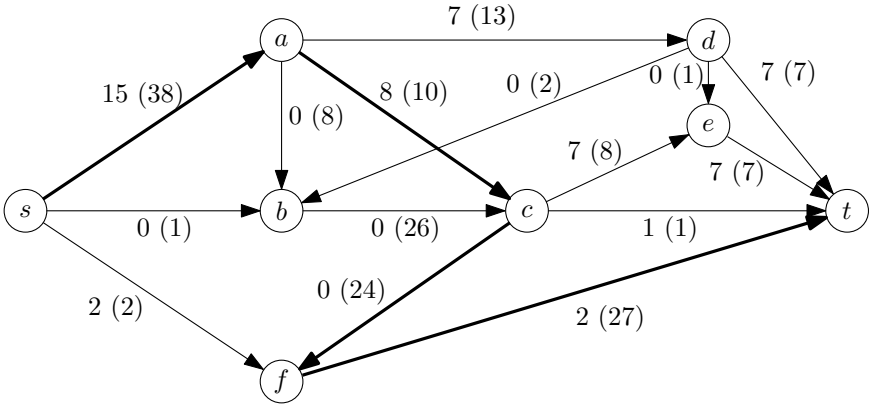
Der nächste Graph zeigt den Fluss nach Auswertung des erhöhenden Weges sowie den neuen gefundenen erhöhenden Weg.

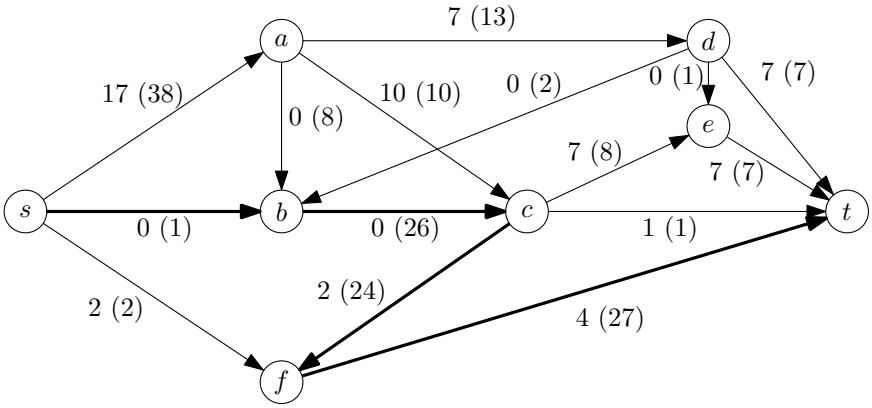


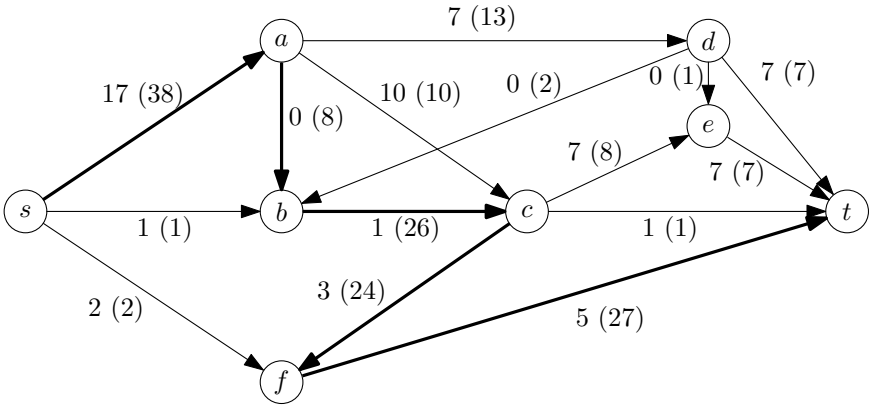

Es folgen die weiteren Zwischenstände des Algorithmus bis zum Ende.

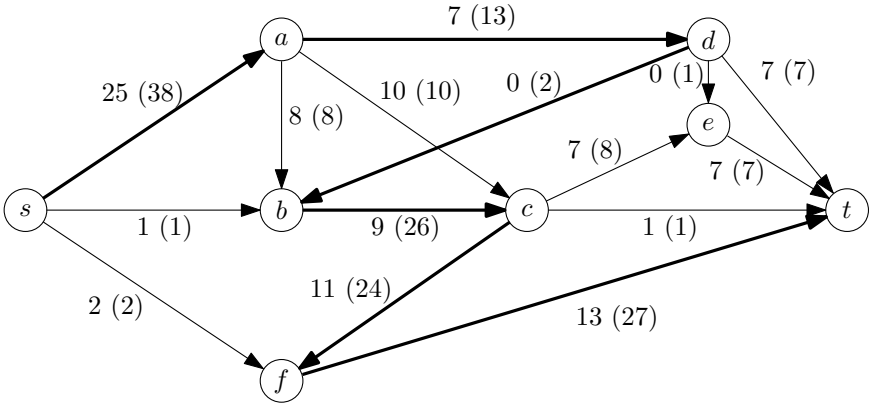

a
d
s
b
c
e
t
f
0 (13)
1 (38)
0 (8)
1 (10)
0 (2)
0 (1)
0 (7)
0 (8)
0 (7)
0 (1)
0 (26)
1 (1)
2 (2)
0 (24)
2 (27)

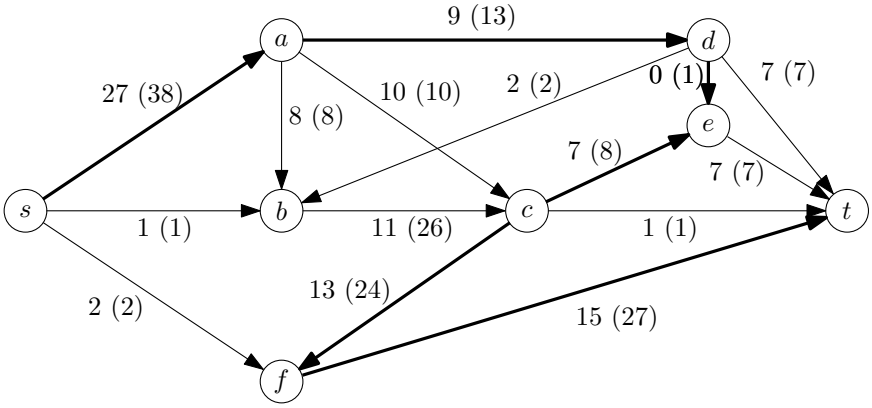

a
d
s
b
c
e
t
f
7 (13)
8 (38)
0 (8)
1 (10)
0 (2)
0 (1)
7 (7)
0 (8)
0 (7)
0 (1)
0 (26)
1 (1)
2 (2)
0 (24)
2 (27)

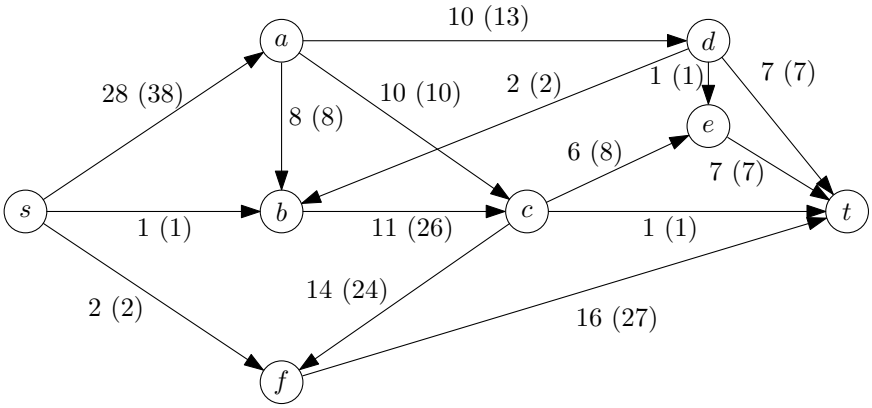

a
d
s
b
c
e
t
f
7 (13)
15 (38)
0 (8)
8 (10)
0 (2)
0 (1)
7 (7)
7 (8)
7 (7)
0 (1)
0 (26)
1 (1)
2 (2)
0 (24)
2 (27)


a
d
s
b
c
e
t
f
7 (13)
17 (38)
0 (8)
10 (10)
0 (2)
0 (1)
7 (7)
7 (8)
7 (7)
0 (1)
0 (26)
1 (1)
2 (2)
2 (24)
4 (27)


a
d
e
s
b
c
t
f
7 (13)
17 (38)
0 (8)
10 (10)
0 (2)
0 (1)
7 (7)
7 (8)
7 (7)
1 (1)
1 (26)
1 (1)
2 (2)
3 (24)
5 (27)


a
d
e
s
b
c
t
f
7 (13)
25 (38)
8 (8)
10 (10)
0 (2)
0 (1)
7 (7)
7 (8)
7 (7)
1 (1)
9 (26)
1 (1)
2 (2)
11 (24)
13 (27)


a
d
e
s
b
c
t
f
9 (13)
27 (38)
8 (8)
10 (10)
2 (2)
0 (1)
7 (7)
7 (8)
7 (7)
1 (1)
11 (26)
1 (1)
2 (2)
13 (24)
15 (27)


a
d
e
s
b
c
t
f
10 (13)
28 (38)
8 (8)
10 (10)
2 (2)
1 (1)
7 (7)
6 (8)
7 (7)
1 (1)
11 (26)
1 (1)
2 (2)
14 (24)
16 (27)

### 4.2.3 Der Algorithmus von Goldberg und Tarjan (1988)

Der Algorithmus von Goldberg und Tarjan ist der *effizienteste* bekannte Algorithmus zur Konstruktion eines Maximalflusses. Er kann unter Benutzung geeigneter Datenstrukturen so implementiert werden, dass er eine Laufzeit von $O(|V|^2|E|^{\frac{1}{2}})$ hat.

Der Algorithmus beruht nicht auf „erhöhenden Wegen“, sondern auf der Verwendung von „Präflüssen“, bei denen für Knoten die Flusserhaltungsbedingung verletzt sein darf: In einem Präfluss darf in einen Knoten mehr hineinfließen als hinausfließt. Der Algorithmus erhält diese „Präfluss-Eigenschaft“. Erst am Ende des Algorithmus wird der Präfluss zu einem Fluss gemacht, der dann maximal ist.

#### Grundidee

Aus Knoten mit „Flussüberschuss“ wird dieser Überschuss in Richtung $t$ geschoben (PUSH). Es werden dazu nicht kürzeste Wege nach $t$, sondern Wege, die momentan „ungefähr“ kürzeste Wege sind, verwendet (RELABEL). Wenn es nicht mehr möglich ist, einen Flussüberschuss in Richtung $t$ zu schieben, so wird er zurück zur Quelle $s$ geschoben.

Zur Vereinfachung der Darstellung erweitern wir das Netzwerk $D = (V, E)$ zu $D' = (V, E')$, wobei $E' := E \cup \{(v, w) \mid (w, v) \in E \wedge (v, w) \notin E\}$. Außerdem wird $c\colon E \to \mathbb{R}_0^+$ fortgesetzt zu $c'\colon V \times V \to \mathbb{R}_0^+$ durch $c'(v, w) = 0$ für $(v, w) \notin E$. Aus Gründen der Übersicht bezeichnen wir im Folgenden allerdings auch $c'$ mit $c$. Ein *Fluss* $f$ ist dann eine Abbildung $f : V \times V \to \mathbb{R}$ mit
*Kapazitätsbedingung*

$$\forall (v, w) \in V \times V \quad f(v, w) \leq c(v, w) \ , \tag{4.3}$$

*Antisymmetrie-Forderung*

$$\forall (v, w) \in V \times V \quad f(v, w) = -f(w, v) \tag{4.4}$$

und *Flusserhaltungsbedingung*

$$\forall v \in V \setminus \{s, t\} \quad \sum_{u \in V} f(u, v) = 0 \ . \tag{4.5}$$

Der *Wert* eines Flusses $f$ ist dann

$$w(f) = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t) \ .$$

Die Antisymmetriebedingung (4.4) bewirkt, dass nicht beide Kanten $(v, w)$ und $(w, v)$ „echten“ Fluss tragen. Dadurch wird die Flusserhaltungsbedingung und die Berechnung des Flusswertes vereinfacht.

**Definition 4.13.** *Ein* Präfluss *ist eine Abbildung $f : V \times V \to \mathbb{R}$, welche die Bedingungen (4.3) und (4.4) erfüllt sowie*

$$\forall v \in V \setminus \{s\} \quad \sum_{u \in V} f(u, v) \geq 0 \ . \tag{4.5'}$$

Die Bedingung (4.5') besagt, dass für alle Knoten $v \in V \setminus \{s\}$ mindestens soviel Fluss hineinfließt wie auch hinausfließt.

**Definition 4.14.** *Sei $f$ ein Präfluss. Für $v \in V \setminus \{t\}$ heißt der Wert*

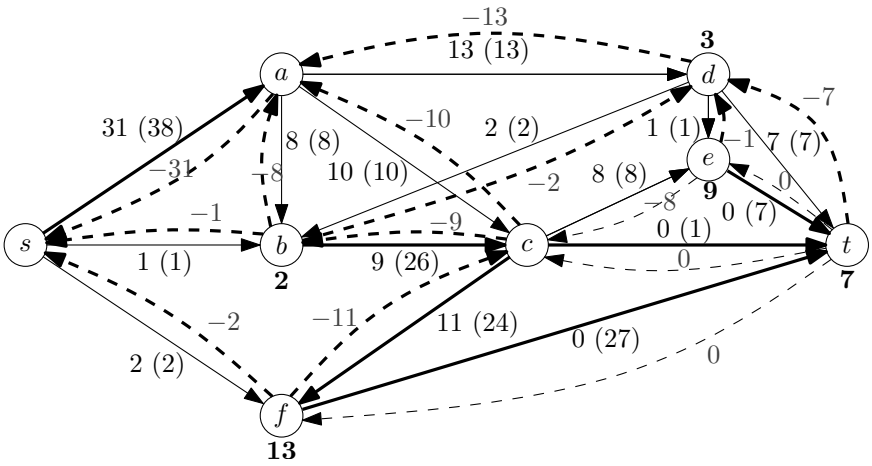
$$e(v) := \sum_{u \in V} f(u, v)$$

Flussüberschuss *und die Abbildung $r_f : E' \to \mathbb{R}$ mit*

$$\forall (u, v) \in E' \quad r_f(u, v) := c(u, v) - f(u, v)$$

*heißt* Restkapazität.

Abbildung 4.1: Flussnetzwerk mit Präfluss




**Bemerkung 4.15.** *Wenn für eine Kante $(u, v) \in E'$ gilt, dass $r_f(u, v) > 0$, so kann der Fluss auf dieser Kante erhöht werden.*

- *Falls $0 \leq f(u, v) < c(u, v)$, so heißt $(u, v)$* nicht saturiert. *Die Erhöhung entspricht dann einer echten Erhöhung auf einer Vorwärtskante.*
- *Ist $0 < f(u, v) \leq c(u, v)$, so ist $(u, v)$* nicht leer, *also $f(v, u) = -f(u, v) < 0 \leq c(v, u)$. Die Erhöhung entspricht dann einer Verringerung des Flusses auf einer Rückwärtskante.*

**Definition 4.16.** *Eine Kante $(v, w) \in E'$ heißt* Residualkante *bezüglich Präfluss $f$, falls $r_f(v, w) > 0$. Der* Residualgraph *zu $f$ ist gegeben durch $D_f(V, E_f)$ mit $E_f := \{(v, w) \in E' \mid r_f(v, w) > 0\}$.*

Eine Kante aus $E$, welche noch nicht saturiert ist, ist eine Residualkante. Eine Kante aus $E' \setminus E$ deren Gegenkante aus $E$ nicht leer ist, ist ebenfalls eine Residualkante.

**Beispiel 4.17.** Die Abbildung 4.1 zeigt in unserem altbekannten Flussnetzwerk einen Präfluss. Die Kapazitäten der Kanten sind in Klammern angegeben, soweit sie ungleich 0 sind. Kanten, die nur in $E'$ sind, sind gestrichelt gezeichnet. Ist in einem Knoten ein Überschuss vorhanden, so ist dieser ebenfalls (in fetter Schrift) angegeben. Residualkanten sind fett hervorgehoben.

**Definition 4.18.** *Eine Abbildung* $\mathrm{dist} : V \to \mathbb{N}_0 \cup \{\infty\}$ *heißt* zulässige Markierung *bezüglich eines Präflusses $f$, falls* $\mathrm{dist}(s) = |V|$, $\mathrm{dist}(t) = 0$, *und für alle $v \in V \setminus \{t\}$, falls $\forall (v, w) \in E_f$,* $\mathrm{dist}(v) \leq \mathrm{dist}(w) + 1$ *gilt. Ein Knoten $v \in V$ heißt* aktiv *im Laufe des Algorithmus, wenn $v \in V \setminus \{t\}$, $e(v) > 0$ und* $\mathrm{dist}(v) < \infty$.

**Bemerkung 4.19.** *Zu Beginn des Algorithmus von Goldberg und Tarjan wird* $\mathrm{dist}(s) := |V|$ *und* $\mathrm{dist}(v) := 0$ *für $v \in V \setminus \{s\}$ gesetzt. Im Laufe des Algorithmus werden die Werte* $\mathrm{dist}(v)$ *verändert, allerdings immer so, dass* dist *zulässig ist. Die Markierung* dist *erfüllt stets:*

- $\mathrm{dist}(s) = |V|$
- *Falls* $\mathrm{dist}(v) < |V|$ *für $v \in V$, so ist* $\mathrm{dist}(v)$ *eine untere Schranke für den Abstand von $v$ zu $t$ in $D_f$ (Residualgraph).*
- *Falls* $\mathrm{dist}(v) > |V|$, *so ist $t$ von $v$ in $D_f$ nicht erreichbar und der Ausdruck* $\mathrm{dist}(v) - |V|$ *ist eine untere Schranke für den Abstand von $v$ zu $s$ in $D_f$.*

Der Algorithmus beginnt mit einem Präfluss $f$ mit $f(s,v) = -f(v,s) = c(s,v)$ für $(s,v),(v,s) \in E'$ und $f(v,w) = 0$ sonst. Der Präfluss $f$ wird im Laufe des Algorithmus verändert, bis er am Ende ein Fluss ist.

### Formale Beschreibung

Der Algorithmus von Goldberg und Tarjan (Algorithmus 35) besteht aus einer Folge von zulässigen Push und Relabel-Operationen auf *aktiven Knoten*.

**Algorithmus 35** : Algorithmus von Goldberg-Tarjan

**Eingabe** : Netzwerk $(D; s, t; c)$ mit $D = (V, E)$ und $c : E \to \mathbb{R}_0^+$
**Ausgabe** : Ein maximaler Fluss $f$

**Für** $(v,w) \in V \times V$ *mit* $(v,w) \notin E$
  $c(v,w) \leftarrow 0$
**Für** $(v,w) \in V \times V$
  $f(v,w) \leftarrow 0$
  $r_f(v,w) \leftarrow c(v,w)$
$\text{dist}(s) \leftarrow |V|$
**Für** $v \in V \setminus \{s\}$
  $f(s,v) \leftarrow c(s,v)$, $r_f(s,v) \leftarrow 0$
  $f(v,s) \leftarrow -c(s,v)$, $r_f(v,s) \leftarrow c(v,s) - f(v,s)$
  $\text{dist}(v) \leftarrow 0$
  $e(v) \leftarrow c(s,v)$
**Solange** *Es gibt einen aktiven Knoten* **tue**
  Wähle einen aktiven Knoten $v$ aus
  Führe für $v$ eine zulässige Operation Push oder Relabel aus
Gib $f$ aus

Die Operation Push (Algorithmus 36) ist zulässig, falls der betreffende Knoten $v$ aktiv ist und falls es ein $w \in V$ mit $r_f(v,w) > 0$ und $\text{dist}(v) = \text{dist}(w) + 1$ gibt.

**Algorithmus 36** : Prozedur Push

**Eingabe** : Tupel $(D, f, v, w)$ aus Netzwerk, Fluss und zwei Knoten
**Seiteneffekte** : Flussüberschuss wird von $v$ nach $w$ über Kante $(v,w)$ geleitet

$\Delta \leftarrow \min\{e(v), r_f(v,w)\}$
$f(v,w) \leftarrow f(v,w) + \Delta$, $f(w,v) \leftarrow f(w,v) - \Delta$
$r_f(v,w) \leftarrow r_f(v,w) - \Delta$, $r_f(w,v) \leftarrow r_f(w,v) + \Delta$
$e(v) \leftarrow e(v) - \Delta$, $e(w) \leftarrow e(w) + \Delta$

Die Operation Relabel (Algorithmus 37) ist zulässig, falls $v$ aktiv ist und falls für alle $w$ mit $r_f(v,w) > 0$ gilt, dass $\text{dist}(v) \leq \text{dist}(w)$.

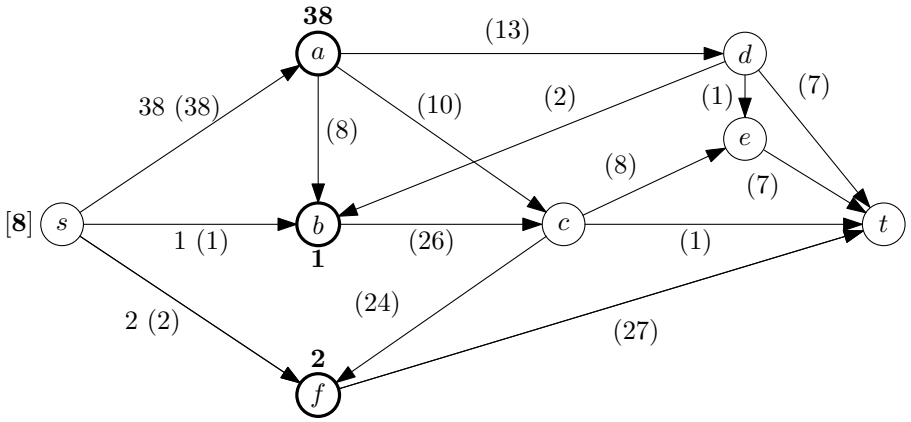
**Algorithmus 37** : Prozedur Relabel

**Eingabe** : Tupel $(D, f, v, \text{dist})$ aus Netzwerk, Fluss, Knoten $v$, und Markierungsfunktion
**Seiteneffekte** : $\text{dist}(v)$ wird erhöht

$$\text{dist}(v) := \begin{cases} \infty \ , & \text{falls } \{w \mid r_f(v,w) > 0\} = \emptyset \ , \\ \min\{\text{dist}(w) + 1 \mid r_f(v,w) > 0\} & \text{sonst.} \end{cases}$$

**Beispiel zu Goldberg-Tarjan**

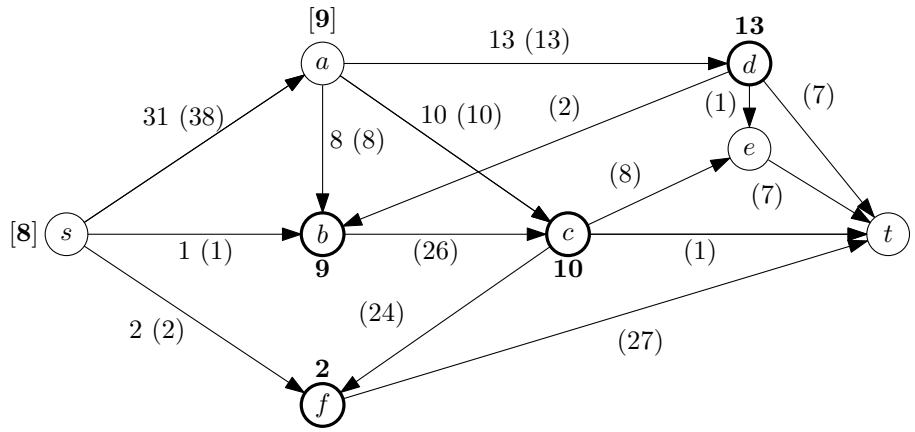
Die Gegenkanten bleiben implizit und werden nicht dargestellt. Die Kapazitäten werden in Klammern dargestellt. Der Präfluss ist nur dann notiert, wenn er von 0 verschieden ist. Für jeden Knoten wird in fetter Schrift die Distanz in eckigen Klammern sowie der Überschuss (jeweils soweit von 0 verschieden) dargestellt. Aktive Knoten werden außerdem fett hervorgehoben. Die erste Darstellung zeigt das Netzwerk zu Beginn des Algorithmus 35.




Wir führen jetzt die folgenden Operationen durch:

- Relabel($a$).
  Dann ist $\text{dist}(a) = 1$.
- Push$(a, b)$ mit $\Delta = 8$ und Push$(a, c)$ mit $\Delta = 10$ und Push$(a, d)$ mit $\Delta = 13$.
  Nach Durchführung dieser drei Operationen ist $e(a) = 7$, $e(b) = 8 + 1 = 9$, $e(c) = 10$ und $e(d) = 13$. Auch der Präfluss hat sich entsprechend verändert.
- Relabel($a$).
  Da $r_f(a, s) > 0$ und $\text{dist}(a) \leq \text{dist}(s)$ ist, folgt $\text{dist}(a) := 9$.
- Push$(a, s)$ mit $\Delta = 7$.
  Danach ist der verringerte Präfluss $f(s, a) = 31$ und $e(a) = 0$.

Nach Durchführung dieser Operationen haben wir den folgenden Zwischenstand im Netzwerk.




Nun folgen diese Operationen:

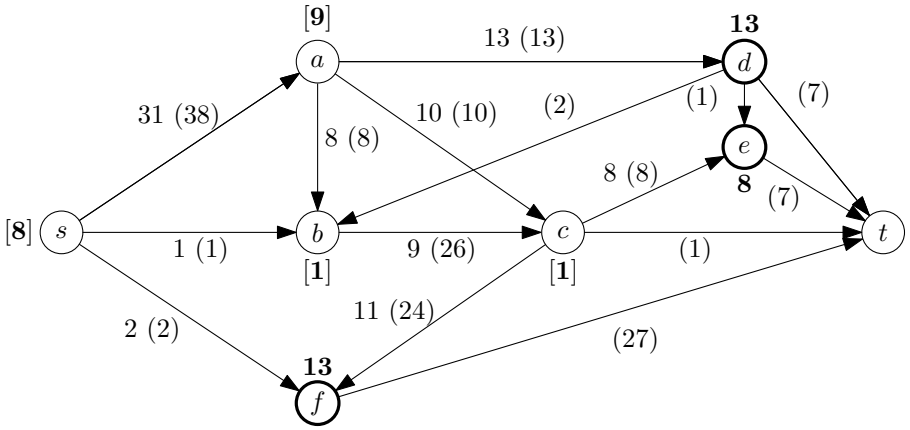
- RELABEL($b$).
  Dann ist dist($b$) = 1.

- PUSH($b, c$) mit $\Delta = 9$.
  Dann ist $e(b) = 0$ und $e(c) = 19$.

- RELABEL($c$).
  Dann ist dist($c$) = 1.

- PUSH($c, e$) mit $\Delta = 8$ sowie PUSH($c, f$) mit $\Delta = 11$.
  Dies hat zur Folge, dass $e(c) = 0$ ist. Dafür haben wir $e(e) = 8$ und $e(f) = 13$.

Der neue Zwischenstand sieht dann so aus:




Weiter:

- RELABEL($d$).
  Dann ist dist($d$) = 1.

- PUSH($d, e$) mit $\Delta = 1$ sowie PUSH($d, t$) mit $\Delta = 7$.
  Nach Durchführung dieser zwei Operationen hat sich $e(d)$ auf 5 verringert, dafür ist jetzt $e(e) = 9$ und $e(t) = 7$.

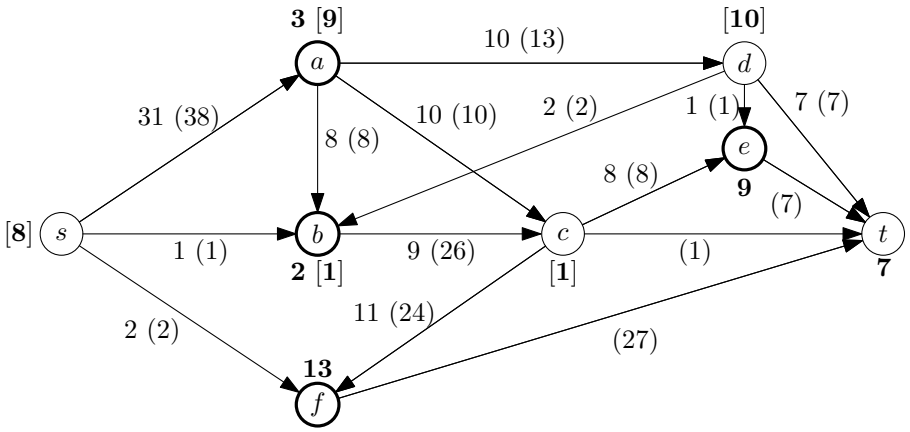
- RELABEL($d$).
  Wegen $r_f(d, b) = 2$ und dist($b$) = 1 folgt dist($d$) = 2.

- PUSH($d, b$) mit $\Delta = 2$.
  Nun ist $e(d) = 3$ und $e(b) = 2$.

- RELABEL($d$).
  Da $r_f(d, a) > 0$, folgt dist($d$) = 10.

- PUSH($d, a$) mit $\Delta = 3$.
  Der Präfluss von $a$ nach $d$ verringert sich auf 10, gleichzeitig ist $e(d) = 0$, dafür jedoch $e(a) = 3$.

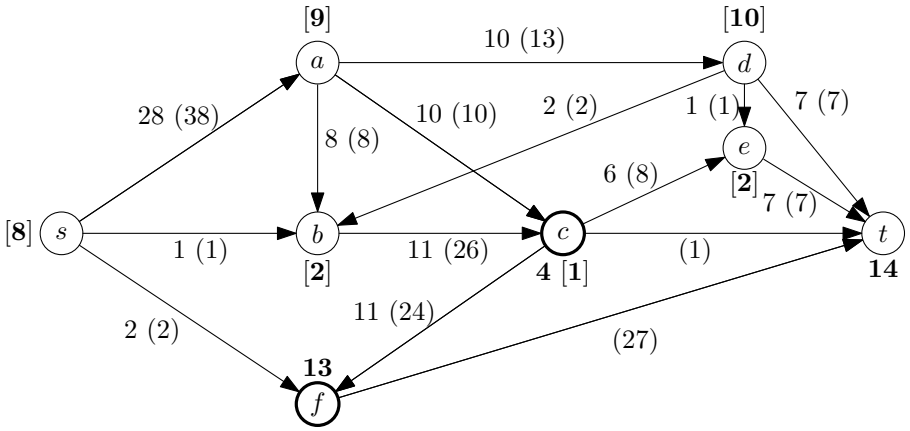
Wir sehen uns erneut den Zwischenstand an:




Nun die folgenden Operationen:

- Push$(a, s)$ mit $\Delta = 3$.
  Der Präfluss $f(s, a)$ verringert sich auf 28 und damit ist $e(a)$ wieder 0.
- Relabel$(b)$.
  Wegen $r_f(b, c) = 17$ und $\text{dist}(c) = 1$ folgt $\text{dist}(b) = 2$.
- Push$(b, c)$ mit $\Delta = 2$.
  Hiermit ist auch $e(b)$ wieder 0, dafür ist $e(c) = 2$.
- Relabel$(e)$, dann Push$(e, t)$ mit $\Delta = 7$.
  Zunächst ist damit $\text{dist}(e) = 1$. Anschließend verringert sich $e(e)$ auf 2, dafür haben wir dann $e(t) = 14$.
- Relabel$(e)$, dann Push$(e, c)$ mit $\Delta = 2$.
  Dann ist $\text{dist}(e) = 2$. Der Präfluss von $c$ nach $e$ verringert sich auf 6, außerdem ist $e(e) = 0$ und $e(c) = 4$.

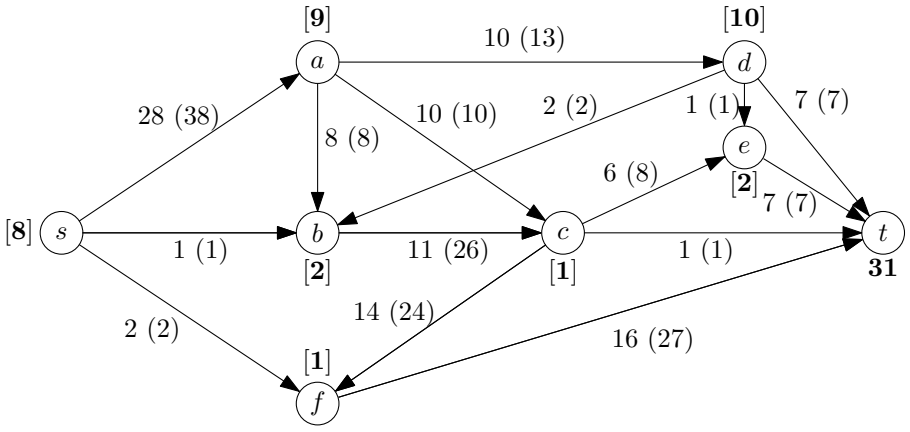
Das Netzwerk sieht nun so aus:




Zum Abschluss noch:

- Push$(c, t)$ mit $\Delta = 1$ und Push$(c, f)$ mit $\Delta = 3$.
  Danach ist $e(c) = 0$, $e(t) = 15$ und $e(f) = 16$.
- Relabel$(f)$, dann Push$(f, t)$ mit $\Delta = 16$.
  Zunächst ist $\text{dist}(f) = 1$. Durch die Push-Operation wird $e(f) = 0$ und $e(t) = 31$.

Nun ist kein Knoten mehr aktiv und ein maximaler Fluss ist gefunden:




**Korrektheit**

Wir beweisen die Korrektheit des Algorithmus von Goldberg und Tarjan, indem wir zunächst zeigen, dass er korrekt ist, falls er terminiert. Dann zeigen wir, dass die maximale Anzahl zulässiger Operationen endlich ist. Dies ist für alle Wahlen aktiver Knoten $v$ und alle Wahlen zulässiger Operationen für $v$ gültig.

**Lemma 4.20.** *Sei $f$ ein Präfluss auf $D$, die Funktion* dist *eine bezüglich $f$ zulässige Markierung auf $V$ und $v \in V$ ein aktiver Knoten. Dann ist entweder eine* Push*-Operation von $v$ oder eine* Relabel*-Operation von $v$ zulässig.*

*Beweis.* Wenn $v$ aktiv ist, so ist $e(v) > 0$ und $\text{dist}(v) < \infty$. Da dist zulässig ist, gilt für alle $w$ mit $r_f(v,w) > 0$, dass $\text{dist}(v) \leq \text{dist}(w) + 1$. Falls nun $\textsc{Push}(v,w)$ für kein solches $w$ zulässig ist, muss $\text{dist}(v) \leq \text{dist}(w)$ sein für alle $w$ mit $r_f(v,w) > 0$. Dann ist aber $\textsc{Relabel}$ zulässig für $v$. □

**Lemma 4.21.** *Während des Ablaufs des Algorithmus von Goldberg und Tarjan ist $f$ stets ein Präfluss und* dist *stets eine bezüglich $f$ zulässige Markierung.*

*Beweis.* Wir führen eine Induktion über die Anzahl $k$ der durchgeführten zulässigen Operationen durch.

Für $k = 0$ ist die Behauptung auf Grund der Initialisierung erfüllt. Wir nehmen also an, die Behauptung gelte nach der $k$-ten Operation und betrachten die $(k+1)$-te Operation.

*Fall 1:* Die $(k+1)$-te Operation ist eine Operation $\textsc{Push}(v,w)$.

Die Präfluss-Eigenschaft von $f$ bleibt erhalten. Die Markierung dist ändert sich nicht, jedoch der Präfluss $f$. Die Operation $\textsc{Push}(v,w)$ erhöht $f(v,w)$ um $\Delta > 0$. Falls dadurch $r_f(v,w) = 0$ wird, bleibt die Markierung dist trivialerweise zulässig. Wird $r_f(w,v) > 0$, so bleibt dist ebenfalls zulässig, da für die Zulässigkeit von $\textsc{Push}(v,w)$ die Bedingung $\text{dist}(w) = \text{dist}(v) - 1 \leq \text{dist}(v) + 1$ gelten muss.

*Fall 2:* Die $(k+1)$-te Operation ist eine Operation $\textsc{Relabel}(v)$.

Dann gilt vor der Operation, dass $\text{dist}(v) \leq \text{dist}(w)$ für alle $w$ mit $r_f(v,w) > 0$. Die Operation $\textsc{Relabel}$ setzt $\text{dist}(v)$ auf das Minimum aller $\text{dist}(w) + 1$ mit $r_f(v,w) > 0$. Danach ist also dist wieder zulässig. Der Präfluss $f$ wird nicht geändert.

**Lemma 4.22.** *Sei $f$ ein Präfluss und* dist *bezüglich $f$ zulässig. Dann ist $t$ im Residualgraph $D_f$ von $s$ aus nicht erreichbar (es gibt also keinen gerichteten $s$-$t$-Weg in $D_f$).*

*Beweis.* Angenommen, es existiert ein einfacher gerichteter $s$-$t$-Weg in $D_f$, etwa

$$s = v_0 \to v_1 \to v_2 \to \cdots \to v_l = t \ .$$

Da dist zulässig ist, gilt $\mathrm{dist}(v_i) \leq \mathrm{dist}(v_{i+1}) + 1$ für $0 \leq i \leq l-1$. Es ist dann also $\mathrm{dist}(s) \leq \mathrm{dist}(t) + l < |V|$, da $\mathrm{dist}(t) = 0$ und $l \leq |V| - 1$. Dies ist ein Widerspruch zu $\mathrm{dist}(s) = |V|$ wegen der Zulässigkeit von dist. □

**Satz 4.23.** *Falls der Algorithmus von Goldberg und Tarjan terminiert und am Ende alle Markierungen endlich sind, dann ist der konstruierte Präfluss ein Maximalfluss im Netzwerk $(D; s, t; c)$.*

*Beweis.* Wegen Lemma 4.20 kann der Algorithmus nur dann abbrechen, wenn kein aktiver Knoten existiert. Da nach Voraussetzung alle Markierungen endlich sind, muss $e(v) = 0$ gelten für alle $v \in V \setminus \{s, t\}$. Damit ist $f$ ein Fluss. Nach Lemma 4.22 gibt es in $D_f$ keinen Weg von $s$ nach $t$. Dann gibt es aber in $D$ keinen bezüglich $f$ erhöhenden $s$-$t$-Weg. □

Es bleibt zu zeigen, dass der Algorithmus terminiert und dass die Markierungen endlich bleiben.

**Lemma 4.24.** *Sei $f$ ein Präfluss auf $D$. Wenn für $v$ gilt, dass $e(v) > 0$, so ist $s$ in $D_f$ von $v$ aus erreichbar.*

*Beweis.* Sei $S_v$ die Menge aller von $v$ in $D_f$ erreichbaren Knoten. Für alle $u \in V \setminus S_v$ und alle $w \in S_v$ ist $f(u, w) \leq 0$, da

$$0 = r_f(w, u) = c(w, u) - f(w, u) \geq 0 + f(u, w) \ .$$

Wegen der Antisymmetriebedingung (4.4) gilt

$$\begin{aligned} \sum_{w \in S_v} e(w) &= \sum_{\substack{u \in V \\ w \in S_v}} f(u, w) \\ &= \sum_{\substack{u \in V \setminus S_v \\ w \in S_v}} f(u, w) + \underbrace{\sum_{u, w \in S_v} f(u, w)}_{=0} \\ &\leq 0 \ . \end{aligned}$$

Da $f$ ein Präfluss ist, ist $e(w) \geq 0$ für alle $w \in V \setminus \{s\}$, also $\sum_{w \in S_v \setminus \{s\}} e(w) \geq 0$. Da aber $e(v) > 0$ und $v \in S_v$, ist auch $s \in S_v$. □

**Lemma 4.25.** *Während des gesamten Algorithmus gilt*

$$\forall v \in V \quad \mathrm{dist}(v) \leq 2|V| - 1 \ .$$

*Beweis.* Zu Beginn des Algorithmus gilt die Behauptung. Betrachten also einen beliebigen Zeitpunkt, zu dem für einen Knoten $v$ die Markierung $\mathrm{dist}(v)$ geändert wird. Dann muss $v$ aktiv sein, also $e(v) > 0$ gelten. Wegen Lemma 4.24 ist $s$ von $v$ aus in $D_f$ erreichbar, es existiert also ein einfacher Weg $v = v_0 \to v_1 \to v_2 \to \cdots \to v_l = s$ mit $\mathrm{dist}(v_i) \leq \mathrm{dist}(v_{i+1}) + 1$ für $0 \leq i \leq l-1$. Da $l \leq |V| - 1$, folgt

$$\mathrm{dist}(v) \leq \mathrm{dist}(s) + l \leq 2|V| - 1 \ .$$ □

**Lemma 4.26.** *Während des Algorithmus werden höchstens $2|V| - 1$ Operationen* Relabel *pro Knoten ausgeführt. Die Gesamtzahl der* Relabel*-Operationen ist also höchstens $2|V|^2$.*

*Beweis.* Jede Operation Relabel an $v$ erhöht $\mathrm{dist}(v)$. Da während des gesamten Algorithmus die Ungleichung $\mathrm{dist}(v) \leq 2|V| - 1$ gilt, folgt die Behauptung. □

Um die Gesamtzahl der PUSH-Operationen abzuschätzen, unterscheiden wir zwei Arten von PUSH-Operationen:

**Definition 4.27.** *Eine Operation* PUSH$(v,w)$ *heißt* saturierend*, wenn hinterher* $r_f(v,w) = 0$ *gilt. Ansonsten heißt* PUSH$(v,w)$ nicht saturierend.

**Lemma 4.28.** *Während des Algorithmus werden höchstens* $2|V||E|$ *saturierende* PUSH *ausgeführt.*

*Beweis.* Ein PUSH$(v,w)$ wird nur für eine Kante $(v,w)$ aus $D_f$ ausgeführt und $\text{dist}(v) = \text{dist}(w)+1$. Falls PUSH$(v,w)$ saturierend ist, so kann nur dann zu einem späteren Zeitpunkt des Algorithmus noch einmal PUSH$(v,w)$ ausgeführt werden, wenn in der Zwischenzeit ein PUSH$(w,v)$ ausgeführt wurde. Dazu muss dann aber $\text{dist}(w) = \text{dist}(v) + 1$ sein, wozu $\text{dist}(w)$ zwischen dem ersten PUSH$(v,w)$ und PUSH$(w,v)$ um mindestens zwei gewachsen sein muss. Ebenso muss vor Ausführung des zweiten PUSH$(v,w)$ auch $\text{dist}(v)$ um mindestens zwei gewachsen sein.

Bei Ausführung der ersten Operation PUSH$(v,w)$ oder PUSH$(w,v)$ überhaupt muss außerdem $\text{dist}(v) + \text{dist}(w) \geq 1$ sein. Am Ende gilt

$$\text{dist}(v) \leq 2|V| - 1 \quad \text{und} \quad \text{dist}(w) \leq 2|V| - 1 \ ,$$

das heißt, dass bei der Ausführung des letzten PUSH$(v,w)$ oder PUSH$(w,v)$ die Ungleichung $\text{dist}(v)+\text{dist}(w) \leq 4|V|-3$ gilt. Für eine Kante $(v,w)$ kann es also höchstens $2|V|-1$ saturierende PUSH-Operationen geben. Damit ist insgesamt $2|V||E|$ eine obere Schranke für die Gesamtzahl der saturierenden PUSH. □

**Lemma 4.29.** *Während des Algorithmus werden höchstens* $4|V|^2|E|$ *nicht saturierende* PUSH *ausgeführt.*

*Beweis.* Wir betrachten die Veränderung des Wertes

$$\mathcal{D} := \sum_{\substack{v \in V \setminus \{s,t\} \\ v \text{ aktiv}}} \text{dist}(v)$$

im Laufe des Algorithmus. Zu Beginn ist $\mathcal{D} = 0$ und es gilt immer $\mathcal{D} \geq 0$.

Jedes nicht saturierende PUSH$(v,w)$ setzt $\mathcal{D}$ um mindestens 1 herab, da danach $e(v) = 0$, also $v$ nicht aktiv und eventuell $w$ danach aktiv, aber $\text{dist}(w) = \text{dist}(v)-1$. Jedes saturierende PUSH$(v,w)$ erhöht $\mathcal{D}$ um höchstens $2|V| - 1$, da der eventuell aktivierte Knoten $w$ nach Lemma 4.25 erfüllt, dass $\text{dist}(w) \leq 2|V| - 1$. Die saturierenden PUSH können also insgesamt (Lemma 4.28) $\mathcal{D}$ um höchstens $(2|V| - 1)(2|V||E|)$ erhöhen. Nach Lemma 4.26 kann durch RELABEL der Wert $\mathcal{D}$ um höchstens $(2|V| - 1)|V|$ erhöht werden.

Da der Gesamtwert der Erhöhungen von $\mathcal{D}$ gleich dem Gesamtwert der Verringerungen von $\mathcal{D}$ ist, haben wir also eine obere Schranke für die Gesamt*zahl* der Verringerungen und damit ist die Gesamtzahl der nicht saturierenden PUSH höchstens

$$(2|V| - 1)(2|V||E| + |V|) \leq 4|V|^2|E| \ . \qquad \square$$

**Satz 4.30.** *Der Algorithmus von Goldberg und Tarjan terminiert nach spätestens* $O(|V|^2|E|)$ *Ausführungen zulässiger* PUSH- *oder* RELABEL-*Operationen.*

### Spezielle Implementationen des Algorithmus von Goldberg und Tarjan

Der Algorithmus von Goldberg und Tarjan ist sehr flexibel. Je nach Wahl der Operationen PUSH oder RELABEL kann man zu unterschiedlichen Worst-case-Laufzeiten kommen. Die Laufzeit ist im wesentlichen abhängig von der Anzahl der nichtsaturierenden PUSH. Dies ist wiederum abhängig von der Wahl des aktiven Knotens. Besonders günstig ist es, aktive Knoten so zu wählen, dass sie nicht „unnötig“ wechseln.

**FIFO-Implementation:** Die aktiven Knoten werden entsprechend der Reihenfolge „first-in-first-out“ gewählt. Dies führt zu $O(|V|^3)$ (Goldberg 1985, Shiloach und Vishkin 1982). Mit „Dynamischen Bäumen“ kommt man sogar mit $O(|V||E| \log \frac{|V|^2}{|E|})$ aus (Goldberg und Tarjan 1988).

**Highest-Label-Implementation:** Für PUSH wird unter den aktiven Knoten derjenige mit höchstem Wert von dist gewählt. Dies führt zu $O(|V|^2|E|^{\frac{1}{2}})$ (Cheriyan und Motvekwani 1989).

**Excess-Scaling-Implementation:** Für PUSH$(v, w)$ wird die Kante $(v, w)$ so gewählt, dass $v$ aktiv, $e(v)$ „geeignet groß“ und $e(w)$ „geeignet klein“ ist. Dies führt zu $O(|E| + |V|^2 \log C)$, wobei $C := \max_{(u,v)} c(u, v)$ ist (Ahuja und Orlin 1989).

### 4.2.4 Anwendungsbeispiel: Mehrmaschinen-Scheduling

Gegeben: Eine Menge von *Aufträgen* (*Jobs*) $j \in J$, $|J| < \infty$ und für jeden Auftrag $j$ eine Bearbeitungszeit $p_j \in \mathbb{R}_0^+$, eine früheste Anfangszeit $r_j \in \mathbb{R}_0^+$ und eine Deadline $d_j \geq r_j + p_j$, außerdem $M$ *Maschinen*. Jede Maschine kann zu einem Zeitpunkt nur einen Job bearbeiten und jeder Job kann zu einem Zeitpunkt nur von einer Maschine bearbeitet werden. Jobs können allerdings unterbrochen werden und später auf derselben oder einer anderen Maschine weiterbearbeitet werden.

Gesucht: Eine Bearbeitungsreihenfolge der Jobs auf den Maschinen, die alle Bedingungen erfüllt, sofern eine solche existiert.

#### Rückführung des Problems auf ein Flussproblem

Die Zeiten $r_j$ und $d_j$ werden für alle $j \in J$ in nichtabsteigender Reihenfolge betrachtet. Es werden die höchstens $2|J| - 1$ paarweise disjunkten Zeitintervalle zwischen diesen Zeitpunkten betrachtet. Diese werden mit $T_{kl}$ für $[k, l)$ bezeichnet. Das Flussnetzwerk wird dann folgendermaßen gebildet:

**Knoten:** Es gibt $|J|$ Knoten für die Jobs, außerdem für jedes $T_{kl}$ einen Knoten und schließlich eine Quelle $s$ und eine Senke $t$.

**Kanten:** Für alle $j \in J$ werden Kanten $(s, j)$ mit $c(s, j) := p_j$ eingeführt, um die Bearbeitungszeiten eines jeden Knotens zu repräsentieren. Außerdem $(T_{kl}, t)$ mit $c(T_{kl}, t) := (l-k)M$, um die Bearbeitungszeit zu repräsentieren, die im Zeitintervall $[k, l)$ zur Verfügung steht, und dann noch $(j, T_{kl})$, falls $r_j \leq k$ und $d_j \geq l$ mit $c(j, T_{kl}) := l - k$, um die maximale Zeit zu repräsentieren, die für Job $j$ im Intervall $[k, l)$ zur Verfügung steht.

Es gibt eine Bearbeitungsreihenfolge, die die Bedingungen erfüllt (*zulässiger Schedule*) genau dann, wenn für einen Maximalfluss $f$ in $D$ gilt, dass $w(f) = \sum_{j \in J} p_j$.

**Beispiel 4.31.** Wir wählen $M := 3$, $J := \{1, 2, 3, 4\}$ und

$$\begin{aligned} p_1 &:= 1.5\ , & r_1 &:= 3\ , & d_1 &:= 5\ , \\ p_2 &:= 1.25\ , & r_2 &:= 1\ , & d_2 &:= 4\ , \\ p_3 &:= 2.1\ , & r_3 &:= 3\ , & d_3 &:= 7\ , \\ p_4 &:= 3.6\ , & r_4 &:= 5\ , & d_4 &:= 9\ . \end{aligned}$$ ■

Daraus ergibt sich, dass wir die Zeitpunkte $1, 3, 4, 5, 7, 9$ betrachten und die Intervalle $T_{13}, T_{34}, T_{45}, T_{57}, T_{79}$. Das zugehörige Netzwerk zeigt dann Abbildung 4.2.

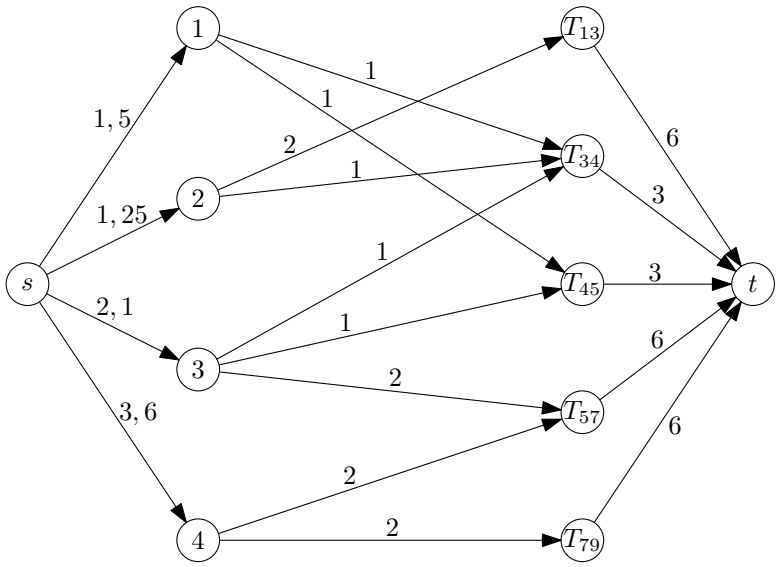



Abbildung 4.2: Flussnetzwerk für Mehrmaschinen-Scheduling

# Kapitel 5

# Kreisbasen minimalen Gewichts

## 5.1 Kreisbasen

> **Der Literaturtip.** Zu diesem Kapitel bieten die Bücher von Deo ([4]) und Bollobás ([1]) einen guten Überblick. Wir werden jedoch im Folgenden mehrmals auf aktuelle Forschungsliteratur verweisen.

Im Kapitel 2 (MST) haben wir eine Knotenfolge $v_1, v_2, \ldots, v_k = v_1, k > 3$ in einem Graph $G = (V, E)$, in der zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind und keine zwei Knoten außer dem Anfangs- und Endknoten mehrfach auftritt, als Kreis bezeichnet. Im Folgenden betrachten wir allgemeinere Kreise und bezeichnen eine solche Knotenfolge als speziellen *einfachen Kreis*.

**Definition 5.1.** *Ein Teilgraph $C = (V_C, E_C)$ von $G = (V, E)$ d.h. $V_c \subseteq V, E_c \subseteq E$ heißt* Kreis *in $G$, falls alle Knoten aus $V_C$ in $C$ geraden Grad haben. Falls $C$ zusammenhängend ist und alle Knoten aus $V_C$ Grad zwei haben, so heißt $C$* einfacher Kreis.

Im Folgenden identifizieren wir einen Kreis $C$ mit seiner Kantenmenge $E_C$. Sei die Kantenmenge $E$ von $G$ beliebig angeordnet, d.h. $E = \{e_1, \ldots e_m\}$. Jede Teilmenge $E' \subseteq E$ kann dann als $m$-dimensionaler Vektor über $\{0, 1\}$ beschrieben werden, so dass für den Vektor $X^{E'}$ *gilt* :

$$X_i^{E'} := \begin{cases} 1, & \text{falls } e_i \in E' \\ 0, & \text{sonst} \end{cases} \tag{5.1}$$

Sei $\mathcal{C}$ die Menge aller Kreise in $G = (V, E)$. Dann induziert $\mathcal{C}$ den Vektorraum der Vektoren $X^c, c \in \mathcal{C}$ über dem Körper GF(2), genannt *Kreisraum* von $G$. Entsprechend induziert die Addition im Kreisraum von $G$ eine Operation $\oplus$ auf $\mathcal{C}$ durch $c_1 \oplus c_2 = (E_{c_1} \cup E_{c_2}) \setminus (E_{c_1} \cap E_{c_2})$. Dies ist die *symmetrische Differenz* der beiden Kantenmengen, sprich die Vereinigung der Kanten abzüglich der gemeinsamen Kanten.

**Bemerkung.** *(i) Die Begriffe* „Dimension des Kreisraums“, „linear unabhängige“ *bzw.* „abhängige“ *Menge von Kreisen sowie der Begriff der* „Kreisbasis“ *ergeben sich in kanonischer Weise. Im Folgenden sei wieder $|V| = n$ und $|E| = m$ für einen Graphen $G = (V, E)$.*

*(ii) Eine Kreisbasis von $G$ kann aus einem aufspannenden Wald $T$ von $G$ bzw. aus einem aufspannenden Baum, falls $G$ zusammenhängend ist, konstruiert werden. O.B.d.A. betrachten wir im Folgenden $G$ als zusammenhängend. Dabei definiert man für jede Nichtbaumkante $e_i = \{v, w\} \in E$ den eindeutigen Kreis $C_i := P(v, w) \cup \{\{v, w\}\}$. Dieser Kreis besteht aus den Kanten von $P(v, w)$, dem eindeutigen einfachen Weg vom Knoten $v$ durch den Baum*

*$T$ zum Knoten $w$, und der Nichtbaumkante $\{v, w\}$. Einen solchen Kreis nennt man* Fundamentalkreis *zu $e_i$. Die Menge all solcher Kreise $C_i$ zu einem aufspannenden Baum $T$ bildet eine Kreisbasis, die man* Fundamentalbasis *zu $T$ nennt.*

*(iii) Die Dimension des Kreisraumes von $G = (V, E)$ ist $m - n + \mathcal{K}(G)$, wobei $\mathcal{K}(G)$ die Anzahl der Zusammenhangskomponenten von $G$ ist (Übung).*

**Bemerkung.** *Sei zu $G = (V, E)$ die Kantengewichtsfunktion $w : E \to \mathbb{R}_0^+$ gegeben. Das Gewicht einer Kreisbasis $\mathcal{B}$ von $G$ sei definiert als $w(\mathcal{B}) := \sum_{C \in \mathcal{B}} w(C) = \sum_{C \in \mathcal{B}} \sum_{e \in C} w(e)$.*

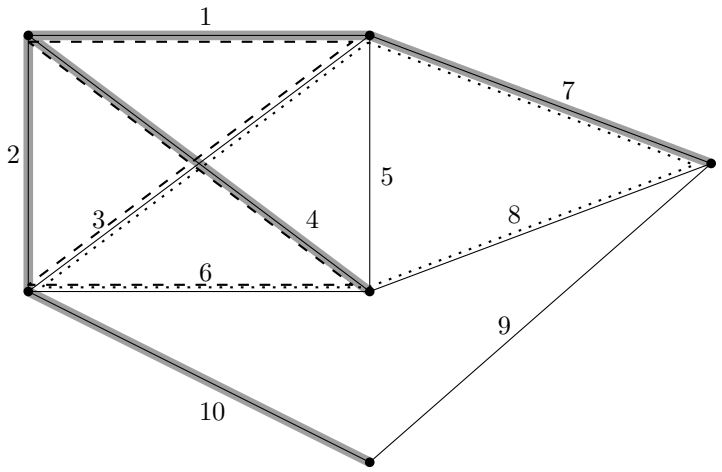
**Problem (MCB).** *Gegeben sei ein Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to \mathbb{R}_0^+$. Finde eine Kreisbasis $\mathcal{B}$ von $G$ mit minimalem Gewicht.*

**Bemerkung.** *(i) Es ist leicht zu sehen, dass jeder Kreis einer MCB einfach ist, vorausgesetzt $w : E \to \mathbb{R}^+$.*

*(ii) Zu jeder Kante $e$ von $G$, die in einem Kreis enthalten ist, gilt: In einer Kreisbasis $\mathcal{B}$ von $G$ gibt es mindestens einen Kreis, der $e$ enthält. Insbesondere enthält jede MCB von $G$ zu jedem $e$ einen in $G$ „kürzesten" Kreis, der $e$ enthält. Im Folgenden nennen wir einen Kreis minimalen Gewichts einfach „kürzester" Kreis.*

*(iii) Die Menge $\mathcal{K} := \{C_{\min}(e_i) : C_{\min}(e_i)$ kürzester Kreis, der $e_i$ enthält, $e_i \in E\}$ ist im Allgemeinen keine Basis (Übung).*

**Beispiel 5.5.** Betrachte folgenden Graphen mit zehn Kanten. Ein Spannbaum, der zur Konstruktion einer Basis verhilft ist fett und grau eingezeichnet.




■

Man findet eine Vielzahl einfacher Kreise:
$1-2-3, 1-4-5, 1-2-6-5, 1-3-6-4, 1-7-8-4, 1-2-6-8-7, 1-7-9-10-2, \ldots, 2-4-6, \ldots, 3-7-8-6, \ldots, 3-5-8-9-10, \ldots$
Zur Konstruktion der Kreisbasis:
Kante 9 induziert $1-7-9-10-2$
Kante 8 induziert $1-7-8-4$
Kante 5 induziert $1-5-4$
Kante 6 induziert $2-4-6$
Kante 3 induziert $1-3-2$
Die Dimension einer Kreisbasis ist $m - n + 1 = 5$, somit bilden diese genannten Kreise eine Kreisbasis.
Basisdarstellung des gestrichelten Kreises: $1-3-6-4 = 1-3-2 \oplus 2-4-6$
Basisdarstellung des gepunkteten Kreises: $3-7-8-6 = 1-3-2 \oplus 1-7-8-4 \oplus 2-4-6$

## 5.2 Das Kreismatroid

Sei $\mathcal{C}$ die Menge aller Kreise in $G = (V, E)$ und $\mathcal{U}$ die Menge aller unabhängigen Teilmengen von $\mathcal{C}$. Dann bildet $(\mathcal{C},\mathcal{U})$ offensichtlich ein Unabhängigkeitssystem. Es gilt sogar der folgende Satz:

**Satz 5.6.** *Das Unabhängigkeitssystem $(\mathcal{C},\mathcal{U})$ ist ein Matroid, genannt* Kreismatroid *von $G$.*

Der Beweis zu Satz 5.6 folgt aus dem Austauschsatz von Steinitz (Übung). Als Folge des Satzes kann eine MCB somit mit Hilfe eines Greedy-Algorithmus wie folgt gefunden werden:

**Algorithmus 38** : MCB-Greedy-Methode

**Eingabe** : Menge $\mathcal{C}$ aller Kreise in $G = (V, E)$.
**Ausgabe** : MCB von $G$

```
1 Sortiere C aufsteigend nach Gewicht zu C_1, ..., C_k
2 B* ← ∅
3 Für i = 1 bis k
4 |   Wenn B* ∪ {C_i} linear unabhängig
5 |   |   B* ← B* ∪ {C_i}
```

Schritt 4, d.h. der Test auf lineare Unabhängigkeit, kann mit Gauss-Elimination durchgeführt werden in $O(k^3), k = |\mathcal{B}^* \cup \{C_i\}| < m$. Allerdings ist die Anzahl der Kreise in einem Graphen im Allgemeinen exponentiell in $m + n$. Die MCB-Greedy-Methode ist also im Allgemeinen nicht polynomial in der Größe von $G$. Eine Idee um zu einem polynomialen Algorithmus für das MCB-Problem zu gelangen, besteht also darin, die Anzahl der Kreise, die in der MCB-Greedy-Methode explizit betrachtet werden, auf eine polynomiale Anzahl zu beschränken.

## 5.3 Der Algorithmus von Horton

**Satz 5.7 (Horton [11]).** *Für jeden Kreis $C$ aus einer MCB von $G$ existiert zu jedem beliebigen Knoten $v$ aus $C$ eine Kante $\{u, w\}$ auf $C$, so dass gilt:*

$$C = \mathrm{SP}(u, v) + \mathrm{SP}(w, v) + \{u, w\},$$

*wobei $\mathrm{SP}(u, v)$ bzw. $\mathrm{SP}(w, v)$ ein kürzester Weg von $u$ bzw. $w$ nach $v$ in $G$ ist.*

Der Beweis zu diesem Satz beruht auf folgendem Lemma über Kreisbasen:

**Lemma 5.8.** *Falls $\mathcal{B}$ eine Kreisbasis ist, $C \in \mathcal{B}$ und $C = C_1 \oplus C_2$, dann ist entweder $\mathcal{B}\setminus\{C\}\cup\{C_1\}$ oder $\mathcal{B} \setminus \{C\} \cup \{C_2\}$ wieder eine Kreisbasis.*

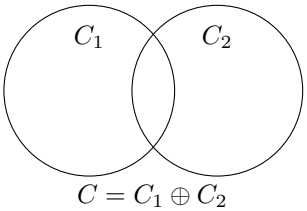



Abbildung 5.1: Elemente der Basis können stets durch ihren Schnitt mit einem anderen Basiselement ersetzt werden

*Beweis.* Siehe Abbildung 5.1 zur Illustration der Aussage und des Beweises. O.B.d.A. sei $C_1 \notin \mathcal{B}$.

(i) Ist der Kreis $C_1$ darstellbar als Linearkombination von Kreisen aus $\mathcal{B} \setminus \{C\}$, so ist offenbar $\mathcal{B} \setminus \{C\} \cup \{C_2\}$ wieder eine Basis.

(ii) Ist der Kreis $C_1$ nur darstellbar als Linearkombination von $C$ und Kreisen aus $\mathcal{B} \setminus \{C\}$, so ist $C_2$ darstellbar als Linearkombination von Kreisen aus $\mathcal{B} \setminus \{C\}$. Somit ist dann $\mathcal{B} \setminus \{C\} \cup \{C_1\}$ wieder eine Basis. □

**Lemma 5.9.** *Sei $\mathcal{B}$ eine Kreisbasis von $G$. Für zwei Knoten $x, y \in V$ und einen Weg $P$ in $G$ von $x$ nach $y$ kann jeder Kreis $C \in \mathcal{B}$, der $x$ und $y$ enthält, ersetzt werden durch einen Kreis $C'$, der $P$ enthält.*

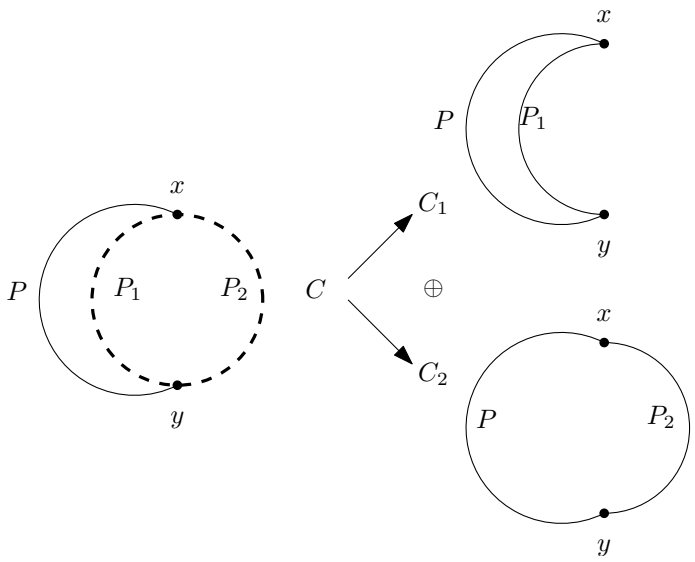



Abbildung 5.2: Elemente der Basis können stets verkleinert werden

*Beweis.* Abbildung 5.2 zeigt die untersuchte Situation auf. Es gilt $C = C_1 \oplus C_2$ und somit nach Lemma 5.8, dass entweder $\mathcal{B} \setminus \{C\} \cup \{C_1\}$ oder $\mathcal{B} \setminus \{C\} \cup \{C_2\}$ wieder eine Basis ist. □

**Folgerung 5.10.** *Angenommen in Abbildung 5.2 ist weder $P_1$ noch $P_2$ ein kürzester Pfad zwischen $x$ und $y$. Sei nun $P$ ein solcher kürzester Pfad, dann gilt offenbar $w(C_1) < w(C)$ und $w(C_2) < w(C)$. Nach Lemma 5.9 kann nun jede Basis $\mathcal{B}$, welche $C$ enthält, in eine Basis $\mathcal{B}'$ umgewandelt werden, welche statt $C$ entweder den Kreis $C_1$ oder $C_2$ enthält, und welche ein geringeres Gewicht hat als $\mathcal{B}$. Falls $\mathcal{B}$ eine MCB ist, so enthält also jeder Kreis in $\mathcal{B}$, der die Knoten $x, y \in V$ enthält, auch einen kürzesten Weg, zwischen $x$ und $y$. Gälte dies nicht, so könnte Lemma 5.9 angewendet werden, um das Gewicht der Basis weiter zu verringern.*

*Beweis (zu Satz 5.7).* Betrachte einen beliebigen Kreis $C$ der MCB, sowie einen beliebigen Knoten $v$ auf $C$. Die Indizierung der Knoten von $C$ sei $v = v_0, \dots v_l = v$. Zum Knoten $v_i$ sei $Q_i$ der Weg auf $C$ von $v$ nach $v_i$, in Richtung der Indizierung der Kreisknoten. Analog sei $P_i$ der Weg auf $C$ von $v_i$ nach $v$, ebenfalls in Richtung der Indizierung der Kreisknoten. Somit teilen $Q_i$ und $P_i$ den Kreis in zwei Hälften, wie in Abbildung 5.3 dargestellt. Dann ist nach Folgerung 5.10 entweder $P_i$ oder $Q_i$ ein kürzester Weg von $v$ nach $v_i$. Teilwege von kürzesten Wegen sind stets kürzeste Wege, sei nun $i$ der größte Index, so dass $Q_i$ der kürzeste Weg von $v$ nach $v_i$ ist. Dann ist $C = Q_i \oplus \{v_i, v_{i+1}\} \oplus P_{i+1}$ die gewünschte Darstellung. □

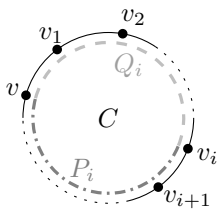



Abbildung 5.3: Darstellung eines Kreises der MCB durch kürzeste Wege

**Algorithmus 39** : Algorithmus von Horton

**Eingabe** : $G = (V, E)$
**Ausgabe** : MCB von $G$

1 $\mathcal{H} \leftarrow \emptyset$
2 **Für** $v \in V$ *und* $\{u, w\} \in E$
3 Berechne $C_v^{uw} = \mathrm{SP}(u, v) + \mathrm{SP}(w, v) + \{u, w\}$
4 **Wenn** $C_v^{uw}$ *einfach ist*
5 $\mathcal{H} \leftarrow \mathcal{H} \cup \{C_v^{uw}\}$
6 Sortiere Elemente aus $\mathcal{H}$ aufsteigend zu $C_1, \ldots, C_k$
7 Wende Schritte 2 bis 5 aus der MCB-Greedy-Methode (Algorithmus 38) an.

Es gilt offensichtlich $|\mathcal{H}| \leq n \cdot m$ und $\mathcal{H}$ kann in $O(n^3)$ berechnet werden, da $m$ in $O(n^2)$ ist. Schritt 6 ist somit in $O(m \cdot n \cdot \log n)$. Die Laufzeit wird also durch die Tests auf lineare Unabhängigkeit dominiert. Dieser Test wird $|\mathcal{H}|$ mal gegen maximal $v = m - n + \mathcal{K}(G)$ Kreise durchgeführt und fällt ebenso oft postitiv aus. Ein einzelner Test erfordert für einen neuen Kreis $O(m^2)$ Operationen. Da $|\mathcal{H}| \leq n \cdot m$ ist, ist die Gesamtlaufzeit also in $O(m^3 \cdot n)$. Wir nennen im Folgenden $\mathcal{H}$ auch *Kandidatenmenge* von Horton.

## 5.4 Der Algorithmus von de Pina (1995)

Sei $T$ ein aufspannender Baum (bzw. Wald) in $G$ und $e_1, \ldots, e_N$ die Nichtbaumkanten aus $G \setminus T$ in einer beliebigen Ordnung, wobei gilt $N = m - n + \mathcal{K}(G)$.

**Algorithmus 40** : Algorithmus von de Pina ([3])

**Eingabe** : Graph $G = (V, E)$
**Ausgabe** : MCB von $G$

1 **Für** $j = 1$ *bis* $N$
2 Initialisiere $S_{1,j} \leftarrow \{e_j\}$
3 **Für** $k = 1$ *bis* $N$
4 Finde einen kürzesten Kreis $C_k$, der eine ungerade Anzahl von Kanten aus $S_{k,k}$ enthält
5 **Für** $j = k + 1$ *bis* $N$
6 $$S_{k+1,j} \leftarrow \begin{cases} S_{k,j} & \text{, falls } C_k \text{ eine gerade Anzahl Kanten aus } S_{k,j} \text{ enthält} \\ S_{k,j} \oplus S_{k,k} & \text{, falls } C_k \text{ eine ungerade Anzahl Kanten aus } S_{k,j} \text{ enthält} \end{cases}$$
7 Ausgabe ist: $\{C_1, \ldots, C_N\}$

Laufzeit: Der Algorithmus von de Pina hat eine Laufzeit in $O(m^3 + m \cdot c)$, wenn in einer Laufzeit von $O(c)$ die Berechnung eines kürzesten Kreises $C_k$ erfolgt, der eine ungerade Anzahl Kanten aus einer vorgegebenen Kantenmenge enthält. Es gilt $c \in O(m^2 + n^2 \log n)$, wobei wir auf einen Beweis für diese Aussage verzichten. Die Gesamtlaufzeit liegt also in $O(m^3 + m \cdot n^2 \log n)$.

### 5.4.1 Beispiel zum Algorithmus von de Pina

Auf den Graphen in Abbildung 5.4 wird nun der Algorithmus von de Pina (Algorithmus 40) angewendet. Offensichtlich ist die Kardinalität einer Basis hier $m - n + 1 = 5$, da der Graph nur eine Zusammenhangskomponente hat.

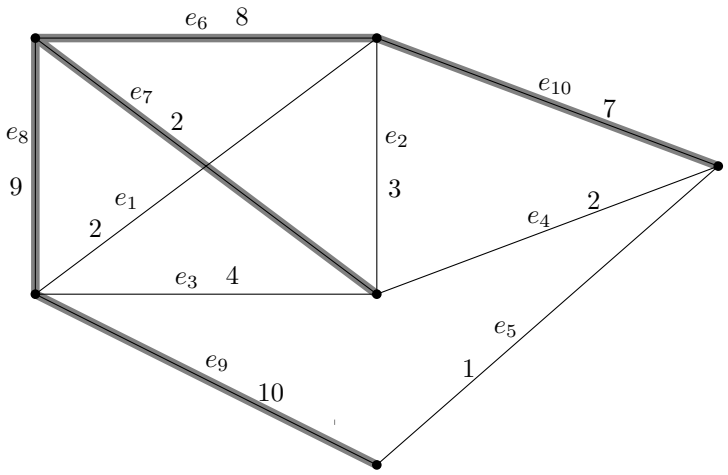



Abbildung 5.4: In diesem Graphen sei der aufspannende Baum $T = \{e_6, e_7, e_8, e_9, e_{10}\}$ gegeben, somit ist $T \setminus G = \{e_1, e_2, e_3, e_4, e_5\}$. Die Kanten haben die eingezeichneten Kantengewichte.

Zunächst erfolgt die Initialisierung der $S_{1,j}$: $S_{1,1} = \{e_1\}$, $S_{1,2} = \{e_2\}$, $S_{1,3} = \{e_3\}$, $S_{1,4} = \{e_4\}$, $S_{1,5} = \{e_5\}$. Anschließend wird über die äußere Schleife iteriert. Es wird in jedem Schritt ein weiterer Kreis in die MCB aufgenommen und alle noch nachfolgenden $S_{i,j}$ zu diesem Kreis „*orthogonalisiert*“. Es resultiert eine Kreisbasis mit Gewicht 66.

$k = 1$: $C_1$ ist ein kürzester Kreis, der eine ungerade Anzahl Kanten aus $S_{1,1}$ enthält. Somit folgt: $C_1 = \{e_1, e_2, e_3\}$ und $w(C_1) = 9$.
$S_{2,2} := S_{1,2} \oplus S_{1,1} = \{e_1, e_2\}$, da $C_1$ eine ungerade Anzahl Kanten (genau $e_2$) mit $S_{1,2}$ gemeinsam hat.
$S_{2,3} := S_{1,3} \oplus S_{1,1} = \{e_1, e_3\}$
$S_{2,4} := \{e_4\}$
$S_{2,5} := \{e_5\}$

$k = 2$: Analog zum ersten Schritt ist nun $C_2$ ein kürzester Kreis, der eine ungerade Anzahl Kanten aus $S_{2,2}$ enthält. Somit folgt: $C_2 = \{e_2, e_4, e_{10}\}$ und $w(C_2) = 12$.
$S_{3,3} = S_{2,3} = \{e_1, e_3\}$
$S_{3,4} = S_{2,2} \oplus S_{2,4} = \{e_1, e_2, e_4\}$
$S_{3,5} = S_{2,5} = \{e_5\}$

$k = 3$: $C_3 = \{e_3, e_7, e_8\}$, und $w(C_3) = 15$
$S_{4,4} = \{e_1, e_2, e_4\}$
$S_{4,5} = \{e_5\}$

$k = 4$: $C_4 = \{e_2, e_6, e_7\}$, und $w(C_4) = 13$
$S_{5,5} = \{e_5\}$

$k = 5$: $C_5 = \{e_5, e_4, e_3, e_9\}$, und $w(C_5) = 17$

### 5.4.2 Korrektheit des Algorithmus von de Pina

Zunächst stellen wir fest, dass Schritt 4 immer einen Kreis liefert. Da $S_{k,k}$ die Kante $e_k$ und keine Kante aus $T$ enthält, gibt es auch einen Kreis $C_k$, der eine ungerade Anzahl Kanten aus $S_{k,k}$ enthält, nämlich der Fundamentalkreis zu $T$ und $e_k$. Dieser Fundamentalkreis besteht aus $e_k$ und dem eindeutigen Weg in $T$ zwischen den Endknoten von $e_k$.

**Problem.** *Warum liefert der Algorithmus von de Pina eine MCB?*

Um zu zeigen, dass der Algorithmus von de Pina tatsächlich eine MCB liefert, betrachten wir eine algebraische Interpretation des Algorithmus, welche im Jahr 2004 von Kavitha, Mehlhorn, Michail und Paluch in [13] präsentiert wurde.

Wir betrachten Kreise wieder als Inzidenzvektoren über $E$, allerdings schränken wir deren Darstellung auf die Menge der Nichtbaumkanten $\{e_1, \ldots, e_N\}$ ein, die restlichen Stellen ergeben sich dann eindeutig über die Fundamentalkreise zu $T$. Die Inzidenzvektoren haben also nicht die volle Dimension $m$ sondern nur die Dimension $m - n + \mathcal{K}(G)$. Als Beispiel betrachte man Kreise im Graphen in Abbildung 5.4, welche nun also über die Nichtbaumkanten $e_1, e_2, e_3, e_4, e_5$ beschrieben werden. Somit entspricht der Kreis $\{e_1, e_2, e_4, e_5, e_9\}$ nun nicht dem Vektor $(1, 1, 0, 1, 1, 0, 0, 0, 1, 0)$ (pro Kante eine Koordinate), sondern dem Vektor $(1, 1, 0, 1, 1)$ (pro Nichtbaumkante eine Koordinate). Daraus kann der Inzidenzvektor über ganz $E$ gebildet werden indem man $C$ mit Hilfe der Fundamentalkreise $C_i$ (zur Kante $e_i$ und dem Baum $T$) rekonstruiert: $C = C_1 \oplus C_2 \oplus C_4 \oplus C_5$.

Wir interpretieren $S_{k,k}$ nach dem $k$-ten Durchlauf von Schritt 3 des Algorithmus von de Pina ebenfalls als Vektor über $\{e_1, \ldots e_N\}$, der Menge der Nichtbaumkanten. Der Vektor $S_{kk}$ bildet jeweils den „*Zeugen*“ dafür, dass der Kreis $C_k$ in der MCB ist. Wir bezeichnen mit $\langle C, S\rangle$ eine Bilinearform der Vektoren $C$ und $S$, nämlich $\langle C, S\rangle = \sum_{i=1}^{N}(c_i \cdot s_i)$, wobei die Summe und auch das Produkt der $c_i$ und $s_i$ in GF(2) berechnet werden. Diese Bilinearform ist zwar identisch mit dem gewöhnlichen inneren Produkt zweier Vektoren, kann aber nicht ohne Weiteres so bezeichnet werden, da sie über GF(2) nicht positiv definit ist, das heißt, aus $\langle x, x\rangle = 0$ folgt nicht notwendigerweise $x = 0$. Da wir diese Eigenschaft jedoch nicht nutzen, sprechen wir im Folgenden, wie bei inneren Produkten üblich, von *Orthogonalität*. Aus der Definition unserer Bilinearform ergibt sich das Folgende:

- Es gilt $\langle C, S\rangle = 0$ genau dann, wenn $C$ und $S$ orthogonal sind.

- Es gilt $\langle C, S\rangle = 1$ genau dann, wenn $C$ eine ungerade Anzahl Einträge mit $S$ gemeinsam hat, also Kreis $C$ eine ungerade Anzahl Nichtbaumkanten aus der Kantenmenge $S$ enthält.

**Beobachtung 5.11.** *Seien $S_1, \ldots, S_k$ linear unabhängige Vektoren, welche den Unterraum $S$ der Dimension $k$ aufspannen. Der zu $S$ orthogonale Raum $R$, ist genau der Lösungsraum des linearen Gleichungssystems $(\langle S_i, X\rangle = 0)_{i=1,\ldots,k}$. Somit hat $R$ die Dimension $N - k$.*

Im Folgenden schreiben wir für $S_{k,k}$ einfach $S_k$. Mit Hilfe der vorangegangenen Formulierungen können wir nun in Algorithmus 41 das algebraische Äquivalent zu Algorithmus 40 notieren.

**Algorithmus 41** : Algorithmus von de Pina algebraisch

**Eingabe** : Graph $G = (V, E)$
**Ausgabe** : MCB von $G$

1 **Für** $i = 1$ *bis* $N$
2 $\quad S_i \leftarrow \{e_i\}$
3 **Für** $k = 1$ *bis* $N$
4 $\quad$ Finde einen kürzesten Kreis $C_k$ mit $\langle C_k, S_k \rangle = 1$
5 $\quad$ **Für** $i = k + 1$ *bis* $N$
6 $\quad\quad$ **Wenn** $\langle C_k, S_i \rangle = 1$
7 $\quad\quad\quad S_i \leftarrow S_i \oplus S_k$
8 Ausgabe ist: $\{C_1, \ldots, C_N\}$

Um Algorithmus 41 zu verstehen, betrachten wir zunächst die Schleife bei Schritt 5. Hier werden für zukünftige Iterationen von Schritt 3 Vektoren $S_k$ „*vorbereitet*“, welche dann Zeuge dafür sind, dass die Kreise $C_k$ in der MCB sind. Die folgende Invariante zeigt, was damit gemeint ist:

**Lemma 5.12.** *Die Schleife bei Schritt 3 von Algorithmus 41 erhält die Invariante* $\langle C_i, S_{j+1} \rangle = 0$ *für alle* $i$, $1 \leq i \leq j \leq N$.

*Beweis.* Wir zeigen per Induktion, über die Durchläufe, dass nach dem $k$-ten Durchlauf der Schleife bei Schritt 3 gilt: $\langle C_i, S_j \rangle = 0$ für alle $i, 1 \leq i \leq k$ und $j, k < j \leq N$. Der Induktionsanfang erfolgt für $k = 1$. Jedes $S_j$, welches noch nicht orthogonal zu $C_1$ ist, wird in Schritt 7 durch die Addition von $S_1$ zu $C_1$ orthogonalisiert, da nach der Addition gilt: $\langle C_1, S_j^{\text{neu}} \rangle = \langle C_1, S_j \oplus S_1 \rangle = \langle C_1, S_j \rangle + \langle C_1, S_1 \rangle = 1 + 1 =_{\text{GF(2)}} 0$.

Sei nun $2 \leq k \leq N$. Angenommen die Behauptung gelte nun für alle Durchläufe vor dem $k$-ten Durchlauf. Betrachte die Kreise $C_1, \ldots, C_k$ bzw. die Zeugen $S_{k+1}, \ldots, S_N$ nach dem $k$-ten Durchlauf. $S_{i,j}$ bezeichne wieder den Zeugen $S_j$ nach dem $i$-ten Durchlauf.

- Falls nun gilt $S_{k,j} = S_{k-1,j}$, so gilt einerseits $\langle C_k, S_{k-1,j} \rangle = 0 = \langle C_k, S_{k,j} \rangle$ wegen des Tests in Schritt 6 und andererseits auch $\langle C_i, S_{k,j} \rangle = 0$ für $i < k$ wegen der Induktionsvoraussetzung.
- Falls $S_{k,j} = S_{k-1,j} \oplus S_{k,k}$ ist, so galt $\langle C_k, S_{k-1,j} \rangle = 1$. Betrachte zunächst $\langle C_i, S_{k,j} \rangle = \langle C_i, S_{k-1,j} \oplus S_{k,k} \rangle = \langle C_i, S_{k-1,j} \rangle + \langle C_i, S_{k,k} \rangle$, für $1 \leq i < k < j \leq N$. Wegen der Induktionsvoraussetzung gilt für den ersten Summanden $\langle C_i, S_{k-1,j} \rangle = 0$ und, da stets $S_{k,k} = S_{k-1,k}$ gilt, ist nach Induktionsvoraussetzung auch $\langle C_i, S_{k,k} \rangle = \langle C_i, S_{k-1,k} \rangle = 0$.

  Letztlich gilt auch noch Orthogonalität mit $C_k$ wegen $\langle C_k, S_{k,j} \rangle = \langle C_k, S_{k-1,j} \rangle + \langle C_k, S_{k,k} \rangle = 1 + 1 =_{\text{GF(2)}} 0$ □

Was insgesamt in Algorithmus 40 und 41 geschieht, kann also beschrieben werden wie in Algorithmus 42.

**Algorithmus 42** : Simple MCB

**Eingabe** : Graph $G = (V, E)$
**Ausgabe** : MCB von $G$

1 $S_1 \leftarrow \{e_1\}$
2 $C_1 \leftarrow$ kürzester Kreis mit $\langle C_1, S_1 \rangle = 1$
3 **Für** $k = 2$ *bis* $N$
4 $\quad$ Berechne beliebigen Vektor $S_k$, der eine nichttriviale Lösung des Systems $\langle C_i, X \rangle = 0$ für $i = 1$ bis $k - 1$ ist
5 $\quad$ Finde einen kürzesten Kreis $C_k$, mit $\langle C_k, S_k \rangle = 1$
6 Ausgabe ist: $\{C_1, \ldots, C_N\}$

**Satz 5.13.** SIMPLE MCB *(Algorithmus 42) berechnet eine MCB.*

*Beweis.* Da jeweils $\langle C_i, S_k\rangle = 0$ für $1 \leq i \leq k-1$ und $\langle C_k, S_k\rangle = 1$, ist $C_k$ linear unabhängig von $\{C_1, \ldots, C_{k-1}\}$. Also ist am Ende $\{C_1, \ldots, C_N\}$ eine Basis. Beachte dabei, dass $N = m - n + \mathcal{K}(G)$ gerade die Dimension des Kreisraumes ist. Es verbleibt zu zeigen, dass $\{C_1, \ldots, C_N\}$ auch minimal ist. Dies zeigen wir per Widerpruchsbeweis.

Angenommen $\{C_1, \ldots, C_N\}$ ist keine MCB, aber $\mathcal{B}$ sei eine MCB. Sei nun $i$ der minimale Index, so dass gilt $\{C_1, \ldots, C_i\} \subseteq \mathcal{B}$, aber für keine MCB $\mathcal{B}'$ gilt $\{C_1, \ldots, C_{i+1}\} \subseteq \mathcal{B}'$. Da $\mathcal{B}$ Basis ist, existieren $D_1, \ldots, D_\ell \in \mathcal{B}$, so dass gilt: $C_{i+1} = D_1 \oplus D_2 \oplus, \ldots, \oplus D_\ell$. Nach Konstruktion ist aber $\langle C_{i+1}, S_{i+1}\rangle = 1$, also existiert ein $D_j, 1 \leq j \leq \ell$ mit $\langle D_j, S_{i+1}\rangle = 1$, und da $C_{i+1}$ kürzester Kreis mit $\langle C_{i+1}, S_{i+1}\rangle = 1$ ist, muss gelten $w(C_{i+1}) \leq w(D_j)$. Setze nun $\mathcal{B}^* := \mathcal{B} \setminus \{D_j\} \cup \{C_{i+1}\}$. Nun ist $\mathcal{B}^*$ wieder Basis (denn es gilt $|\mathcal{B}^*| = N$ und alle Elemente sind linear unabhängig) und es gilt $w(\mathcal{B}^*) \leq w(\mathcal{B})$, also ist $\mathcal{B}^*$ sogar eine MCB. Da $\langle D_j, S_{i+1}\rangle = 1$ ist und $\langle C_j, S_{i+1}\rangle = 0$ für $1 \leq j \leq i$ ist $D_j \notin \{C_1, \ldots, C_i\}$. Also ist $\mathcal{B}^*$ eine MCB mit $\{C_1, \ldots, C_i, C_{i+1}\} \subseteq \mathcal{B}^*$, im Widerspruch zur Wahl von $i$. □

**Bemerkung.** *(i) Die Laufzeit kann auf $O(m^2 \cdot n + m \cdot n^2 \log n)$ reduziert werden (siehe [13]).*

*(ii) Empirisch kann die Laufzeit verbessert werden, indem man Horton und de Pina „verheiratet". Um den Aufwand zur Berechnung des $C_k$ mit $\langle C_k, S_k\rangle = 1$ zu reduzieren, wird in Schritt 5 in Algorithmus 42 nur der kürzeste Kreis $C_k$ mit $\langle C_k, S_k\rangle = 1$ aus der Kandidatenmenge von Horton bestimmt, d.h. der Lösungsraum wird eingeschränkt.*

*(iii) Der Algorithmus von Horton kann mittels schneller Matrix-Multiplikation auf eine Laufzeit von $O(m^\omega \cdot n)$ reduziert werden, wobei $\omega$ der Exponent für die Matrix-Multiplikation ist. Bekannt ist, dass gilt $\omega < 2,376$.*

## 5.5 Effiziente Berechnung eines Zertifikats für MCB

Es ist im Allgemeinen wünschenswert zu einem Algorithmus einen effizienten Testalgorithmus zur Verfügung zu haben, der zu einer Eingabe und der entsprechenden Ausgabe ein Zertifikat dafür liefert, dass der Algorithmus tatsächlich das Gewünschte liefert. Der Testalgorithmus sollte eine wesentlich geringere Laufzeit als der Algorithmus selbst haben.

**Problem (Zertifikat für MCB-Algorithmus A).** *Gegeben sei der Graph $G = (V, E), w : E \to \mathbb{R}_0^+$ und eine Menge von Kreisen $\mathcal{A}$ von $G$. Gib ein Zertifikat dafür an, dass $\mathcal{A}$ eine MCB von $G$ ist.*

**Bemerkung 5.15.** *Zur Ausgabe eines Algorithmus kann vorab leicht überprüft werden, dass die Ausgabe tatsächliche eine Menge von Kreisen ist und ob deren Anzahl gleich der Dimension des Kreisraums ist.*

**Algorithmus 43** : MCB-CHECKER

**Eingabe** : Graph $G = (V, E)$, Kreise $C_1, \ldots C_N$
**Ausgabe** : Zertifikat zur Prüfung, ob $C_1, \ldots C_N$ eine MCB von $G$ sind
1 Berechne aufspannenden Wald $T$, dabei seien $\{e_1, \ldots, e_N\}$ die Nichtbaumkanten
2 Definiere $A := (C_1 \ldots C_N)$ als $N \times N$-Matrix, deren $i$-te Spalte der Inzidenzvektor von $C_i$ mit $\{e_1, \ldots, e_N\}$ ist
3 Berechne $A^{-1}$

Die Matrix $A$ ist genau dann invertierbar, wenn die Kreise $C_1, \ldots, C_N$ linear unabhängig sind, d.h. eine Kreisbasis bilden. Die Zeilen $S_1, \ldots, S_N$ von $A^{-1}$ bilden das Zertifikat, dass $C_1, \ldots, C_N$ eine MCB bilden. Das folgende Lemma zeigt, dass mit Hilfe des Zertifikats leicht geprüft werden kann, ob $C_1, \ldots, C_N$ eine MCB bilden.

**Lemma 5.16.** *Seien $S_1, \dots, S_N$ und $C_1, \dots, C_N$ linear unabhängig. Falls $C_i$ ein kürzester Kreis mit $\langle S_i, C_i \rangle = 1$ für alle $1 \leq i \leq N$ ist, dann ist $C_1, \dots, C_N$ eine MCB.*

**Bemerkung 5.17.** *Allgemeiner kann man sogar beweisen, dass für die Matrix mit Spalten $A_1 \dots A_N$, wobei $A_i$ jeweils ein kürzester Kreis mit $\langle S_i, A_i \rangle = 1$ ist, stets gilt:*

$$\sum_{i=1}^{N} w(A_i) \leq w(\mathcal{B})$$

*für jede Kreisbasis $\mathcal{B}$.*

*Beweis.* Betrachte die Permutation $\Pi$ auf $1, \dots, N$ so, dass gilt $w(A_{\Pi(1)}) \leq w(A_{\Pi(2)}) \leq \dots \leq w(A_{\Pi(N)})$. Seien $\{C_1^*, \dots, C_N^*\}$ eine MCB mit $w(C_1^*) \leq, \dots, \leq w(C_N^*)$. Wir zeigen nun, dass $w(A_{\Pi(i)}) \leq w(C_i^*)$ für alle $1 \leq i \leq N$, daraus folgt dann die Behauptung.

Zunächst gibt es ein $k$ und ein $\ell$ mit $1 \leq k \leq i \leq \ell \leq N$, so dass $\langle C_k^*, S_{\Pi(\ell)} \rangle = 1$, ansonsten wären alle $N - i + 1$ linear unabhängigen Vektoren $S_{\Pi(i)}, \dots, S_{\Pi(N)}$ orthogonal zu $C_1^*, \dots, C_i^*$. Der Teilraum orthogonal zu $C_1^*, \dots, C_i^*$ hat jedoch nach Beobachtung 5.11 nur die Dimension $N - i$. Da $A_{\Pi(\ell)}$ ein kürzester Kreis mit $\langle A_{\Pi(\ell)}, S_{\Pi(\ell)} \rangle = 1$ ist, gilt $w(A_{\Pi(\ell)}) \leq w(C_k^*)$. Außerdem ist $w(A_{\Pi(i)}) \leq w(A_{\Pi(\ell)})$ und $w(C_k^*) \leq w(C_i^*)$. Da nun $w(A_{\Pi(i)}) \leq w(C_i^*)$ für alle $1 \leq i \leq N$ gilt, folgt auch die Behauptung $\sum_{i=1}^{N} w(A_i) \leq w(\mathcal{B})$. □

Wenn $A = (C_1 \dots C_N)$ invertierbar ist, so sind die Zeilen $S_1 \dots S_N$ von $A^{-1}$ und auch $C_1 \dots C_N$ linear unabhängig und $\langle S_i, C_i \rangle = 1$ für alle $1 \leq i \leq N$. Wenn also $C_i$ außerdem ein kürzester Kreis mit $\langle S_i, C_i \rangle = 1$ ist, so ist $\{C_1, \dots, C_N\}$, nach Lemma 5.16 eine MCB. Wenn andererseits für ein $i$ mit $1 \leq i \leq N$ der Kreis $C_i$ nicht ein kürzester Kreis mit $\langle S_i, C_i \rangle = 1$ wäre, dann ließe sich wie im Korrektheitsbeweis zu SIMPLE MCB (Algorithmus 42) der Kreis $C_i$ durch einen kürzeren Kreis $C_i$ mit $\langle S_i, C_i \rangle = 1$ ersetzen, ohne dass die lineare Unabhängigkeit verletzt werden würde. Insgesamt ist also $\{C_1, \dots, C_N\}$ eine MCB genau dann, wenn für alle $i$ mit $1 \leq i \leq N$ der Kreis $C_i$ ein kürzester Kreis mit $\langle S_i, C_i \rangle = 1$ ist.

# Kapitel 6

# Lineare Programmierung

Ein Ansatz zur Lösung von Optimierungsproblemen besteht darin, das Problem als ein *lineares Programm* (LP) zu formulieren und dann einen allgemeinen Algorithmus zur Lösung von LPs anzuwenden.

**Problem (Standardform eines LP).** *Gegeben seien eine Matrix $A \in \mathbb{R}^{m \times n}$ und Vektoren $b \in \mathbb{R}^m, c \in \mathbb{R}^n$. Bestimme $x \in \mathbb{R}^n$ so, dass die Zielfunktion*

$$\sum_{i=1}^{n} c_i \cdot x_i = c^T x$$

*minimiert wird unter den Nebenbedingungen*

$$Ax \geq b \quad \textit{und} \quad x \geq 0,$$

*beziehungsweise so, dass die Zielfunktion*

$$\sum_{i=1}^{n} c_i \cdot x_i = c^T x$$

*maximiert wird unter den Nebenbedingungen*

$$Ax \leq b \quad \textit{und} \quad x \geq 0.$$

**Bemerkung.** *(i) Durch Vorzeichenwechsel kann ein Minimierungsproblem trivialerweise in ein Maximierungsproblem transformiert werden (und umgekehrt), das heißt $c^T x$ wird ersetzt durch $-c^T x$.*

*(ii) Programme, die nicht in Standardform sind, lassen sich mehr oder weniger leicht in Standardform bringen:*

- *Gleichungen der Form $\sum_{j=1}^{n} a_{ij} x_j = b_i$ werden ersetzt durch zwei Ungleichungen $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ und $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$.*
- *Ungleichungen der Form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ werden ersetzt durch $\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i$.*
- *Eine möglicherweise negative Variable $x_i \in \mathbb{R}$ wird ersetzt durch den Ausdruck $x_i' - x_i''$ für zwei Variablen $x_i' \geq 0$ und $x_i'' \geq 0$.*

Grundlegende Fragestellungen der linearen Programmierung bestehen darin, zu einem gegebenen LP zu bestimmen

- ob es eine zulässige Lösung gibt, d.h. ein $x^*$ das die Nebenbedingungen erfüllt;

- ob es eine optimale Lösung gibt, beziehungsweise wie eine optimale Lösung aussieht.

Oft ist gefordert, dass $x$ nur ganzzahlige Einträge hat (oder sogar nur Einträge aus $\{0,1\}$). In diesem Fall ist die Bestimmung einer optimalen Lösung NP-schwer (siehe Informatik III).

**Beispiel 6.2 (Lieferproblem).** Rohöl soll durch ein chemisches und/oder ein physikalisches Verfahren in Komponenten zerlegt werden:

- schweres Öl $S$
- mittelschweres Öl $M$
- leichtes Öl $L$

Es stehen zwei mögliche Verfahren zur Auswahl:

1. Verfahren
   10 Einheiten Rohöl ergeben
   $\left.\begin{array}{lll} 2 & \text{Einheiten} & S \\ 2 & \text{Einheiten} & M \\ 1 & \text{Einheit} & L \end{array}\right\}$ Kosten dafür seien 3.

2. Verfahren
   10 Einheiten Rohöl ergeben
   $\left.\begin{array}{lll} 1 & \text{Einheiten} & S \\ 2 & \text{Einheiten} & M \\ 4 & \text{Einheit} & L \end{array}\right\}$ Kosten dafür seien 5.

Es sei eine Lieferverpflichtung von
$\left.\begin{array}{lll} 3 & \text{Einheiten} & S \\ 5 & \text{Einheiten} & M \\ 4 & \text{Einheit} & L \end{array}\right\}$ zu erfüllen.
Diese Lieferverpflichtung soll so kostengünstig wie möglich erfüllt werden. Wir führen die Variablen $x_1$ und $x_2$ für die Verfahren 1 und 2 ein. Das LP lautet dann:

$$\min 3 \cdot x_1 + 5 \cdot x_2$$

unter den Nebenbedingungen

$$\begin{aligned} 2{\cdot}x_1 + \quad x_2 &\geq 3 && S\text{-Ungleichungen} \\ 2{\cdot}x_1 + 2{\cdot}x_2 &\geq 5 && M\text{-Ungleichungen} \\ x_1 + 4{\cdot}x_2 &\geq 4 && L\text{-Ungleichungen} \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

■

**Beispiel 6.3 (Flussproblem).** Gegeben ist ein Flussnetzwerk bestehend aus dem gerichteten Graphen $D = (V, E)$, Kantenkapazitäten $c : E \to \mathbb{R}_0^+$, und der Quelle $\in V$ und der Senke $t \in V$. Ist $(p, q)$ eine Kante so bezeichnen wir mit $x_{p,q}$ den Fluss auf dieser Kante. Das LP für einen maximalen Fluss lautet:

$$\max \sum_{(s,i)\in E} x_{s,i} - \sum_{(i,s)\in E} x_{i,s}$$

$$\text{unter den Nebenbedingungen} \quad \left.\begin{array}{l} x_{i,j} \leq c_{i,j} \\ x_{i,j} \geq 0 \end{array}\right\} \quad \forall (i,j) \in E$$

$$\sum_{j:(i,j)\in E} x_{i,j} - \sum_{j:(j,i)\in E} x_{j,i} = 0 \quad \forall j \in V \setminus \{s,t\}$$

. ∎

Eine grundlegende Tatsache in Theorie und Praxis der Lineare Optimierung besteht darin, dass es zu jedem Standardprogramm ein eindeutiges nichttriviales *duales Programm* gibt. Sei:

$$\begin{aligned} P: \quad & \min c^T x \quad \text{unter} \\ & Ax \geq b \quad \text{und} \quad x \geq 0 \end{aligned}$$

$$c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$$

das *primale Programm*. Dann ist das zugehörige duale Programm:

$$\begin{aligned} D: \quad & \max y^T b \quad \text{unter} \\ & y^T A \leq c^T \quad \text{und} \quad y \geq 0 \end{aligned}$$

$$y \in \mathbb{R}^m$$

**Satz 6.4 (Schwacher Dualitätssatz).** *Seien zu $P$ und $D$ Vektoren $x_0 \in \mathbb{R}^n$ und $y_0 \in \mathbb{R}^m$ mit $Ax_0 \geq b, x_0 \geq 0$ und $y_0^T A \leq c^T, y_0 \geq 0$ gegeben. Dann gilt:*

$$y_0^T b \leq c^T x_0$$

*Beweis.*

$$y_0^T b \leq y_0^T (Ax_0) = (y_0^T A) x_0 \leq c^T x_0$$

**Bemerkung 6.5.** *Es gilt sogar für Optimallösungen $x^*$ und $y^*$ zu $P$ beziehungsweise $D$, dass $y^{*T} b = c^T x^*$. Dazu später mehr.*

## 6.1 Geometrische Repräsentation von LPs

**Definition (Grundlegende Begriffe).** *(i) Eine Menge $P \subseteq \mathbb{R}^n$ heißt* konvex*, falls für alle $s,t \in P$, $0 < \lambda < 1$ auch die* Konvexkombination $\lambda \cdot s + (1-\lambda) \cdot t$ *in $P$ ist.*

*(ii) Ein Punkt $p \in P$, $P$ konvexe Menge, heißt* Extrempunkt*, falls es kein Punktepaar $s,t \in P$ gibt, mit $p = \lambda \cdot s + (1-\lambda) \cdot t, 0 < \lambda < 1$.*

*(iii) Die* konvexe Hülle *von $P \subseteq \mathbb{R}^n$ ist die kleinste konvexe Menge $P' \subseteq \mathbb{R}^n$ mit $P \subseteq P'$. Man sagt dann: $P$ spannt $P'$ auf.*

*(iv) Eine Menge $S := \{s \in \mathbb{R}^n : a^T \cdot s \leq \lambda\}$ für $a \in \mathbb{R}^n, \lambda \in \mathbb{R}$ heißt* Halbraum*.*

**Beobachtung 6.7.** *Da jeder Halbraum konvex ist, ist auch der Schnitt endlich vieler Halbräume konvex. Ein $P \subseteq \mathbb{R}^n$, das der Schnitt endlich vieler Halbräume ist, heißt auch* konvexes Polyeder*. Seine Extrempunkte nennt man* Ecken*.*

**Beobachtung 6.8.** *Ein (konvexes) Polyeder ist nicht notwendigerweise beschränkt. Falls $P \subseteq \mathbb{R}^n$ der Schnitt endlich vieler Halbräume und beschränkt ist, d.h. falls es eine endliche Punktemenge $P' \subseteq \mathbb{R}^n$ gibt, so dass $P$ von $P'$ aufgespannt wird, so heißt $P$* konvexes Polytop*.*

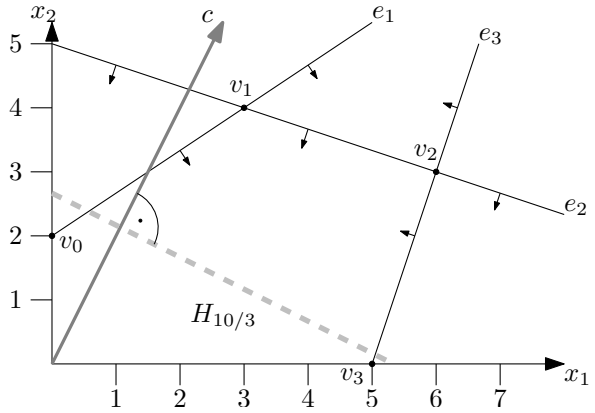



Abbildung 6.1: Die $e_i$ definieren die Halbräume. Die Hyperebene $H_{10/3}$ zeigt alle Punkte, wo die Optimierungsfunktion den Wert 10/3 annimmt. Punkte $v_0, v_1, v_2, v_3$ sind Extrempunkte, $v_1$ und $v_2$ sind benachbart.

Wir können LPs also geometrisch darstellen:

- Die Nebenbedingungen $a_i^T x \leq b_i$ beziehungsweise $a_i^T x \geq b_i$ ($a_i$ ist die $i$-te Zeile von $A$) definieren jeweils einen Halbraum. Die Grenze dieses Halbraums ist die *Hyperebene* $a_i^T x = b_i$.

$$\begin{aligned} -2{\cdot}x_1 + 3{\cdot}x_2 &\leq 6 \quad (e_1) \qquad x_1 \geq 0 \qquad \max x_1 + 2 \cdot x_2 \quad (c) \\ x_1 + 3{\cdot}x_2 &\leq 15 \quad (e_2) \qquad x_2 \geq 0 \\ 3{\cdot}x_1 - \quad x_2 &\leq 15 \quad (e_3) \end{aligned}$$

  Entsprechend bilden die zulässigen Lösungen eines LP ein konvexes Polyeder. Im Folgenden betrachten wir nur LPs, deren Menge von zulässigen Lösungen nicht leer ist.

- Die Zielfunktion $c^T x$ gibt eine Richtung im $\mathbb{R}^n$ an und kann durch den Richtungsvektor $c$ und Hyperebene $H_z = \{x \in \mathbb{R}^n : c^T x = z\}$, $z \in \mathbb{R}$, die zu $c$ orthogonal sind, visualisiert werden. Nenne $z$ den *Zielwert* zu $H_z$ und $c$ *Zielvektor*.

  **Bemerkung 6.9.** *Das konvexe Polyeder der zulässigen Lösungen eines LP ist nicht notwendigerweise beschränkt. Damit das LP eine optimale Lösung besitzt, muss es auch nur in Richtung des Zielvektors beschränkt sein.*

  Ein LP, dessen Zielwert $c^T x$ in Richtung des Zielvektors durch die Nebenbedingungen beschränkt ist, heißt *beschränktes LP*.

- Die Extrempunkte des konvexen Polyeders zu einem LP sind jeweils Schnittpunkte von $n$ Hyperebenen zu linear unabhängigen Nebenbedingungen. Zwei Extrempunkte heißen *benachbart*, wenn sie sich in genau einer Hyperebene unterscheiden. Benachbarte Extrempunkte sind durch eine *Kante* verbunden, die gerade den Schnitt der $n-1$ gemeinsamen Hyperebenen entspricht.

**Definition 6.10 (Oberhalb, unterhalb).** *Zu einem Ursprungsvektor $c$ sei $H^c$, die (zu $c$ orthogonale) Hyperebene, definiert durch $H^c := \{x \mid x^T c = |c|^2\}$. Ein Punkt $v$ liegt genau dann* oberhalb *$H^c$, wenn $v^T c > |c|^2$. Analog werden die Begriffe* unterhalb, nach oben *und* nach unten *definiert.*

**Bemerkung 6.11.** *Sei $P$ ein konvexes Polyeder zu einem beschränkten LP mit der Zielfunktion $c^T x$, und sei $H_2$ eine zu $c$ orthogonale Hyperebene, die $P$ schneidet. Verschiebt man $H_2$ entlang $c$*

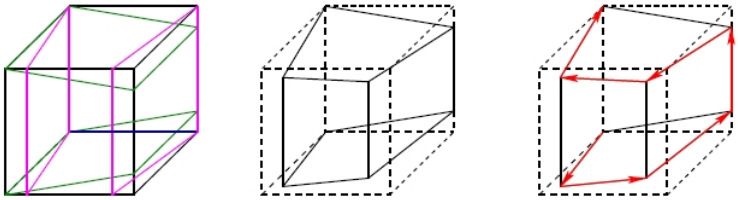


Abbildung 6.2: Konstruiert man Ebenen, die jeweils etwas schief zu den Seiten eines Würfels liegen, so liefert die eingezeichnete Eckenfolge eine streng monoton wachsende Zielfunktion. Dennoch läuft diese Eckenfolge jede der $2^3$ Ecken des entstandenen Polyeders ab. Dieses Beispiel, der Klee-Minty-Polyeder, ist in höhere Dimensionen skalierbar und liefert einen exponentiellen Aufwand.

*(d.h. ändert man $z$ entsprechend), so erhöht bzw. verringert sich der Zielwert entsprechend. Verschiebt man nun die Hyperebene soweit, bis sich zum ersten Mal kein Punkt von $P$ mehr „oberhalb" der Hyperebene befindet, dann sei $H^*$ die so erhaltene Hyperebene. Auf diese Weise erhält man mit $H^* \cap P$ die optimalen Lösungen des LPs. Wegen der Konvexität von $P$ ist mindestens einer dieser Punkte ein Extrempunkt von $P$.*

**Folgerung 6.12.** *Für ein beschränktes LP gibt es eine Optimallösung, deren Wert einem Extrempunkt des konvexen Polyeders zum LP entspricht.*

Sei nun o.B.d.A. ein Maximierungsproblem gegeben. Sei zudem $v$ ein beliebiger Extrempunkt des Polyeders $P$, $H(v)$ die zu $c$ orthogonale Hyperebene, die durch $v$ verläuft, und $e$ eine zu $v$ inzidente Kante. Es ist nun eine *lokale Charakterisierung* eines Extrempunktes zum Optimalwert wie folgt möglich:

1. Falls $e$ oberhalb von $H(v)$ verläuft, so verbessert sich der Zielwert, wenn man von $v$ startend entlang $e$ läuft.
2. Falls $e$ unterhalb von $H(v)$ verläuft, so verschlechtert sich der Zielwert, wenn man von $v$ startend entlang $e$ läuft.

Im Fall 1 nennt man $e$ *verbessernde Kante*. Falls $v$ keine verbessernde Kante hat, können wir $H(v)$ nicht nach oben verschieben ohne $P$ zu verlassen (auf Grund der Konvexität von $P$), das heißt $v$ ist eine Optimallösung.

**Satz 6.13.** *Für jedes beschränkte LP in Standardform mit Lösungspolyeder $P$ gibt es mindestens einen Extrempunkt von $P$, der den optimalen Zielwert annimmt. Ein Extrempunkt des Lösungspolyeders ist genau dann optimal, wenn es keine von diesem Punkt ausgehende verbessernde Kante gibt.*

## 6.2 Algorithmen zur Lösung von LPs

Die *Simplexmethode* (Danzig, 1951) beruht auf der von uns hergeleiteten Vorgehensweise. Sie ist im Worst-case im Allgemeinen exponentiell, funktioniert in der Praxis aber recht gut. Die Suche nach der nächsten anzusteuernden Ecke ist der Kern des Simplexalgorithmus. Selbst wenn der Eckenwechsel stets eine Verbesserung der zu optimierenden Funktion garantieren soll, können *alle* Ecken abgelaufen werden bevor das Optimum erreicht wird, siehe Abbildung 6.2 für ein Beispiel, das von Klee und Minty (1972) stammt. Obwohl es randomisierte Strategien zur Pivotsuche gibt, welche eine polynomiale erwartete Laufzeit haben, und sogar deterministische Strategien mit einer polynomialen Average-case Laufzeit, ist noch immer nicht bekannt, ob es eine Strategie zur Pivotsuche mit polynomialer Worst-case Laufzeit gibt.

Der erste polynomiale Algorithmus ist die 1979 von Khachian vorgestellte *Ellipsoidmethode*. In der Theorie war dieses Verfahren ein Durchbruch, denn dies war der Beweis, dass die Problemklasse LP in $\mathcal{P}$ liegt. In der Praxis ist das Verfahren jedoch nicht so gut wie die Simplexmethode.

Im Jahre 1984 wurde von Kharmakar die *Innere-Punkte-Methode* vorgestellt, die ebenfalls polynomiale Laufzeit hat und in einigen praktischen Fällen auch gut funktioniert. Im Allgemeinen ist aber auch diese Methode weniger praktikabel als die Simplexmethode.

Bei vielen Problemstellungen muss eine sinnvolle Lösung eines linearen Programms ganzzahlig sein, solch ein Problem heißt *Integer Lineares Programm* oder ILP. Zwar gibt es (nicht-triviale) ILPs, die in Polynomialzeit berechnet werden können, wie zum Beispiel Matching- und ganzzahlige Flussprobleme, im Allgemeinen jedoch sind ILPs $\mathcal{NP}$-schwer, wie beispielsweise Knapsack. Eine häufig angewendete Technik zum Lösen von ILPs ist es, das Programm zu *relaxieren*, indem man zunächst keine ganzzahlige Lösung fordert und dann das so entstandene LP löst, um anschließend mit Hilfe von Heuristiken aus der Lösung des LPs eine „*gute*“ Lösung des ILPs zu konstruieren.

# Kapitel 7

# Approximationsalgorithmen

In diesem Kapitel wollen wir $\mathcal{NP}$-schwere Optimierungsprobleme untersuchen. Da es unwahrscheinlich ist, dass es für solche Probleme polynomiale Algorithmen gibt, die eine optimale Lösung berechnen, wollen wir polynomiale Approximationsalgorithmen entwerfen.

Bezeichne $\Pi$ ein Optimierungsproblem und $I$ eine Instanz zu $\Pi$. Sei $\mathcal{A}$ ein Approximationsalgorithmus für $\Pi$, d.h. ein Algorithmus, der zu jeder Instanz $I$ von $\Pi$ eine zulässige, aber nicht notwendigerweise optimale Lösung berechnet.

- $\mathcal{A}(I)$ bezeichne den Wert der Lösung, die $\mathcal{A}$ für $I$ berechnet.
- OPT(I) bezeichne den Wert einer optimalen Lösung für $I$.

$\mathcal{A}$(I) heißt *absoluter Approximationsalgorithmus*, falls es ein $K \in \mathbb{N}_0$ gibt, so dass gilt:

$$|\mathcal{A}(I) - \mathrm{OPT}(I)| \leq K \qquad \text{für alle Instanzen } I \text{ von } \Pi.$$

Es gibt nur wenige $\mathcal{NP}$-schwere Optimierungsprobleme, für die es polynomiale, absolute Approximierungsalgorithmen gibt. Häufiger bekannt sind Negativ-Aussagen folgender Form: Falls $\mathcal{P} \neq \mathcal{NP}$, so kann es keinen polynomialen, absoluten Approximationsalgorithmus zu $\Pi$ geben.

**Problem (Knapsack).** *Gegeben sei eine Menge $M$ von Objekten. Seien*

$$\omega : M \longrightarrow \mathbb{N}_0 \qquad \textit{und} \qquad c : M \longrightarrow \mathbb{N}_0$$

*Funktionen und $W \in \mathbb{N}_0$. Finde $M' \subseteq M$ mit*

$$\sum_{a \in M} c(a) \textit{ maximal} \qquad \textit{und} \qquad \sum_{a \in M'} \omega(a) \leq W.$$

Das Knapsack-Problem ist $\mathcal{NP}$-schwer (es ist sogar $\mathcal{NP}$-vollständig).

**Satz 7.1.** *Falls $\mathcal{P} \neq \mathcal{NP}$, so gibt es keinen (polynomialen) absoluten Approximationsalgorithmus $\mathcal{A}$ für* Knapsack *mit*

$$|\mathcal{A}(I) - OPT(I)| \leq K \qquad \textit{für alle } I \textit{ von } \textsc{Knapsack} \textit{ und festes } K \in \mathbb{N}_0.$$

*Beweis.* Angenommen es gäbe einen solchen Algorithmus $\mathcal{A}$. Betrachte eine beliebige Instanz $I$ mit $M$, $w\colon M \to \mathbb{N}_0$, $c\colon M \to \mathbb{N}_0$ und $W \in \mathbb{N}_0$. Betrachte dazu Instanz $I'$ mit $M$, $w$, $W$ und $c'\colon M \to \mathbb{N}_0$ definiert durch

$$c'(x) := (K+1) \cdot c(x).$$

Für jedes $I'$ liefert $\mathcal{A}$ den Wert $\mathcal{A}(I')$ mit

$$|\mathcal{A}(I') - \mathrm{OPT}(I')| \leq K.$$

$\mathcal{A}(I')$ induziert einen Lösung $M^*$ für $I$ mit Wert $c(M^*)$, für die gilt:

$$|(K+1) \cdot c(M^*) - (K+1) \cdot \mathrm{OPT}(I)| \leq K.$$

Also ist

$$|c(M^*) - \mathrm{OPT}(I)| \leq \frac{K}{K+1} < 1.$$

Wegen $\mathrm{OPT}(I) \in \mathbb{N}$ könnte man somit KNAPSACK in polynomialer Zeit lösen, dies ist ein Widerspruch zu $\mathcal{P} \neq \mathcal{NP}$. □

## 7.1 Approximationsalgorithmen mit relativer Gütegarantie

Ein polynomialer Algorihmus $\mathcal{A}$, der für ein vorgegebenes Optimierungsproblem $\Pi$ für jede Instanz $I$ von $\Pi$ einen Wert $\mathcal{A}(I)$ liefert mit

$$\mathcal{R}_{\mathcal{A}}(I) \leq K, \text{wobei}$$

$$\mathcal{R}_{\mathcal{A}}(I) \quad := \quad \begin{cases} \frac{\mathcal{A}(I)}{\mathrm{OPT}(I)} & \text{, falls } \Pi \text{ Minimierungsproblem} \\ \\ \frac{\mathrm{OPT}(I)}{\mathcal{A}(I)} & \text{, falls } \Pi \text{ Maximierungsproblem} \end{cases}$$

und $K$ Konstante ($K \geq 1$), heißt *absoluter Approximationsalgorithmus* mit *relativer Gütegarantie* oder auch *relativer Approximationsalgorithmus.*

Zu einem relativen Approximationsalgorithmus zu $\Pi$ definiere

$$\mathcal{R}_{\mathcal{A}} \quad := \quad \inf\{r \geq 1 : \ \mathcal{R}_{\mathcal{A}}(I) \leq r \quad \text{für alle } I \text{ von } \Pi\}.$$

$\mathcal{A}$ heißt *$\epsilon$-approximierender Algorithmus* falls $\mathcal{R}_{\mathcal{A}} \leq 1 + \epsilon$.

### 7.1.1 Das allgemeine Knapsack Problem

**Problem (Allgemeines KNAPSACK).** *Gegeben sei eine Menge von $n$ Objekten. Seien $\omega_1, \ldots, \omega_n \in \mathbb{N}$, $c_1, \ldots, c_n \in \mathbb{N}_0$ und $W, C \in \mathbb{N}_0$. Existieren $x_1, \ldots, x_n \in \mathbb{N}_0$ mit folgenden Eigenschaften*

$$\sum_{i=1}^{n} x_i w_i \leq W \qquad \textit{und} \qquad \sum_{i=1}^{n} x_i c_i \geq C \ ?$$

Das bedeutet, dass von jedem Exemplar mehrere „eingepackt“ werden können. Dieses Problem ist natürlich auch $\mathcal{NP}$-vollständig.

**Beispiel 7.2.** Algorithmus 44 zeigt einen greedy-KNAPSACK-Algorithmus für die Optimierungsversion des Allgemeinen KNAPSACK-Problems ($\max \sum_{i=1}^{n} x_i c_i$). Im Schritt 1 von Algorithmus 44 wird nach den Kostendichten sortiert. An der Beschreibung des Algorithmus sieht man, dass so viele Exemplare der aktuellen Kostendichteklasse wie möglich eingepackt werden. Die Laufzeit des Greedy-KNAPSACK-Algorithmus wird offensichtlich durch das Sortieren der $p_i$ in Schritt 1 dominiert. Da Sortieren in $O(n \log n)$ möglich ist, ergibt sich für diesen Algorithmus eine Laufzeit von $O(n \log n)$. ■

**Algorithmus 44** : GREEDY-KNAPSACK

**Eingabe** : $\omega_1, \ldots, \omega_n \in \mathbb{N}$, $c_1, \ldots, c_n \in \mathbb{N}_0$ und $W \in \mathbb{N}_0$
**Ausgabe** : Gute Lösung zum allgemeinen KNAPSACK Problem (Wert der Lösung)

1 Berechne die „Kostendichten“ $p_i := \frac{c_i}{w_i}$ und indiziere diese so, dass gilt $p_1 \geq p_2 \geq \ \ldots \ \geq p_n$ für $i = 1, \ \ldots \ , n$.
2 **Für** $i = 1$ *bis* $n$
3 $\quad x_i \leftarrow \left\lfloor \frac{W}{w_i} \right\rfloor$
4 $\quad W \leftarrow W - x_i \cdot w_i$

**Satz 7.3.** *Der Greedy-*KNAPSACK*-Algorithmus $\mathcal{A}$ erfüllt $\mathcal{R}_{\mathcal{A}} = 2$.*

*Beweis.* Ohne Einschränkung sei $w_1 \leq W$. Es gilt offensichtlich:

$$\mathcal{A}(I) \geq c_1 \cdot \left\lfloor \frac{W}{w_1} \right\rfloor \qquad \text{für alle } I$$

Weiterhin gilt für die optimale Lösung $\mathrm{OPT}(I)$:

$$\mathrm{OPT}(I) \leq c_1 \cdot \left( \left\lfloor \frac{W}{w_1} \right\rfloor + 1 \right)$$

Wäre das nicht so, also wäre $\mathrm{OPT}(I) > c_1 \cdot \left( \left\lfloor \frac{W}{w_1} \right\rfloor + 1 \right)$, dann müsste wegen der Zulässigkeit von $\mathrm{OPT}(I)$ und wegen $p_1 = \max\{\frac{c_i}{w_i}\}$ gelten: $W > w_1 \cdot \left( \left\lfloor \frac{W}{w_1} \right\rfloor + 1 \right) \geq W$, ein Widerspruch. Somit gilt:

$$\begin{aligned} \mathrm{OPT}(I) \leq \quad & c_1 \cdot \left( \left\lfloor \frac{W}{w_1} \right\rfloor + 1 \right) \\ & \leq 2 \cdot c_1 \cdot \left\lfloor \frac{W}{w_1} \right\rfloor \\ & \leq 2 \cdot \mathcal{A}(I), \end{aligned}$$

also $\mathcal{R}_{\mathcal{A}} \leq 2$.

Behandle nun folgendes spezielle KNAPSACK-Beispiel: Sei $n = 2$, $c_1 = 2 \cdot w_1$, $c_2 = 2 \cdot w_2 - 1$, $w_2 = w_1 - 1$ und $W = 2 \cdot w_2$. Offensichtlich ist nun:

$$\frac{c_1}{w_1} = \frac{2 \cdot w_1}{w_1} = 2 > \frac{c_2}{w_2} = \frac{2 \cdot w_2 - 1}{w_2} = \frac{2 \cdot w_1 - 3}{w_1 - 1}.$$

Für hinreichend großes $W$ verwendet der Greedy-Algorithmus also ausschließlich $w_1$, und zwar genau einmal, wegen $W = 2 \cdot w_2 = 2 \cdot w_1 - 2$. Er findet somit die Lösung $\mathcal{A}(I) = c_1 = 2 \cdot w_1$. Die optimale Lösung ist es jedoch, ausschließlich $w_2$ zu verwenden, und zwar genau zweimal ($W = 2 \cdot w_2$), somit gilt $\mathrm{OPT}(I) = 2 \cdot c_2 = 4 \cdot w_2 - 2$. Somit folgt für die Approximationsgüte

$$\frac{\mathrm{OPT}(I)}{\mathcal{A}(I)} = \frac{2 \cdot w_1 - 3}{w_1} \xrightarrow{w_1 \to \infty} 2$$

d.h. für dieses Greedy-Verfahren gilt $\mathcal{R}_{\mathcal{A}} \geq 2$. Damit ist also $R_{\mathcal{A}} = 2$. □

### 7.1.2 Bin Packing (Optimierungsversion)

**Problem.** *Gegeben sei eine endliche Menge $M = \{a_1, \ldots, a_n\}$ mit der Gewichtsfunktion*

$$s \colon M \longrightarrow (0, 1].$$

*Gesucht ist eine Zerlegung von $M$ in eine minimale Anzahl von Teilmengen $B_1, \ldots, B_m$, so dass*

$$\sum_{a_i \in B_j} s(a_i) \leq 1 \qquad \text{für } 1 \leq j \leq m.$$

Das Entscheidungsproblem Bin Packing ist äquivalent zu dem Problem Multiprocessor Scheduling und daher $\mathcal{NP}$-vollständig. Im Folgenden betrachten wir einige Approximationsalgorithmen für Bin Packing, als erstes Next Fit (Algorithmus 45). Die Laufzeit ist in $\mathcal{O}$(n).

**Algorithmus 45** : Next Fit (NF)

**Eingabe** : Menge $M$ und Gewichtsfunktion $s$
**Ausgabe** : Approximationslösung für Bin Packing

```
1 Füge a_1 in B_1 ein
2 Für a_ℓ ∈ {a_2, ... a_n}
3     Sei B_j die letzte, nicht-leere Menge
4     Wenn s(a_ℓ) ≤ 1 − Σ_{a_i ∈ B_j} s(a_i)
5         Füge a_ℓ in B_j ein
6     sonst
7         Füge a_ℓ in B_{j+1} ein
```

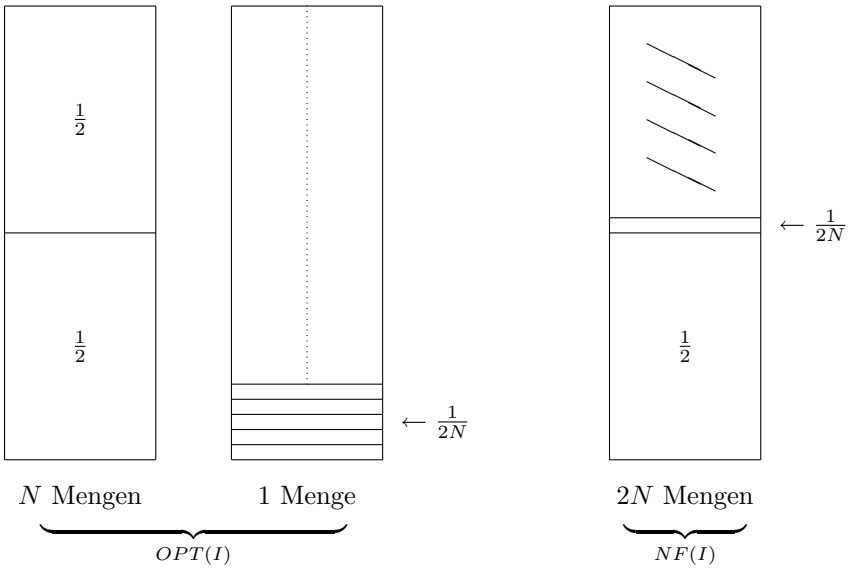
**Beispiel 7.4.** Sei $n = 4 \cdot N$ für ein $N \in \mathbb{N}$. Sei weiterhin

$$s(a_i) = \begin{cases} \frac{1}{2} & i \text{ ungerade} \\ \frac{1}{2 \cdot N} & \text{sonst} \end{cases}$$

Der Next Fit Algorithmus $\mathcal{A}_{NF}$ benötigt $2 \cdot N$ Mengen $B_j$, während eine Optimallösung mit $N+1$ Mengen auskommt, das bedeutet

$$\mathcal{A}_{NF}(I) = 2 \cdot \mathrm{OPT}(I) - 2.$$

■




**Satz 7.5.** Next Fit *erfüllt* $\mathcal{R}_{NF} = 2$.

*Beweis.* Sei $I$ eine Instanz von Bin Packing mit $\mathcal{A}_{NF}(I) = k$ und $B_1, \ldots, B_k$ seien die benutzten Mengen. Sei für $1 \leq j \leq k$

$$s(B_j) = \sum_{a_i \in B_j} s(a_i)$$

Dann gilt folgende Ungleichung für $j = 1, \ldots, k-1$:

$$s(B_j) + s(B_{j+1}) > 1,$$

da ansonsten die Elemente aus $B_{j+1}$ von NF in $B_j$ eingefügt worden wären. Daraus folgt

$$\sum_{j=1}^{k} s(B_j) \quad > \quad \tfrac{k}{2} \qquad \text{falls } k \text{ gerade, beziehungsweise}$$

$$\sum_{j=1}^{k-1} s(B_j) \quad > \quad \tfrac{k-1}{2} \qquad \text{falls } k \text{ ungerade.}$$

Also ist $\mathrm{OPT}(I) > \frac{k-1}{2}$, also $k = \mathrm{NF}(I) < 2 \cdot \mathrm{OPT}(I) + 1$. Da $\mathrm{NF}(I)$ allerdings ganzzahlig ist, gilt somit sofort $\mathrm{NF}(I) \leq 2 \cdot \mathrm{OPT}(I)$. □

**Bemerkung 7.6.** *Für Beispiele mit $s(a_i) \leq s \leq \frac{1}{2}$ für $1 \leq i \leq n$ kann dieses Resultat noch verschärft werden. Für jede Menge $B_j$ muss dann $s(B_j) > 1 - s$ sein, außer möglicherweise für $B_k$, $k = \mathcal{A}_{NF}(I)$. Dann folgt:*

$$OPT(I) \geq \sum_{j=1}^{k-1} s(B_j) > (k-1) \cdot (1-s),$$

$$\textit{also} \qquad k = \mathcal{A}_{NF}(I) < \frac{1}{1-s} \cdot OPT(I) + 1.$$

*Für $s \longrightarrow 0$ geht also $NF(I) \longrightarrow OPT(I) + 1$.*

**Algorithmus 46** : First Fit (FF)

**Eingabe** : Menge $M$ und Gewichtsfunktion $s$
**Ausgabe** : Approximationslösung für Bin Packing

```
1 Füge a_1 in B_1 ein
2 Für a_ℓ ∈ {a_2, ... a_n}
3 |  j ← max{i : B_1, ..., B_i nicht leer}
4 |  r ← min{t ≤ j + 1 : s(a_ℓ) ≤ 1 − Σ_{a_i ∈ B_t} s(a_i)}
5 |  Füge a_ℓ in B_r ein
```

Die Laufzeit ist in $\mathcal{O}(n^2)$. Offensichtlich ist $\mathrm{FF}(I) < 2 \cdot \mathrm{OPT}(I)$ für alle Instanzen I, denn es kann nach dem Anwenden von First Fit höchstens eine Menge $B_j$ geben mit $s(B_j) \leq \frac{1}{2}$. Also gilt:

$$\mathrm{FF}(I) < 2 \cdot \sum_j s(B_j) = 2 \cdot \sum_{a_i \in M} s(a_i)$$

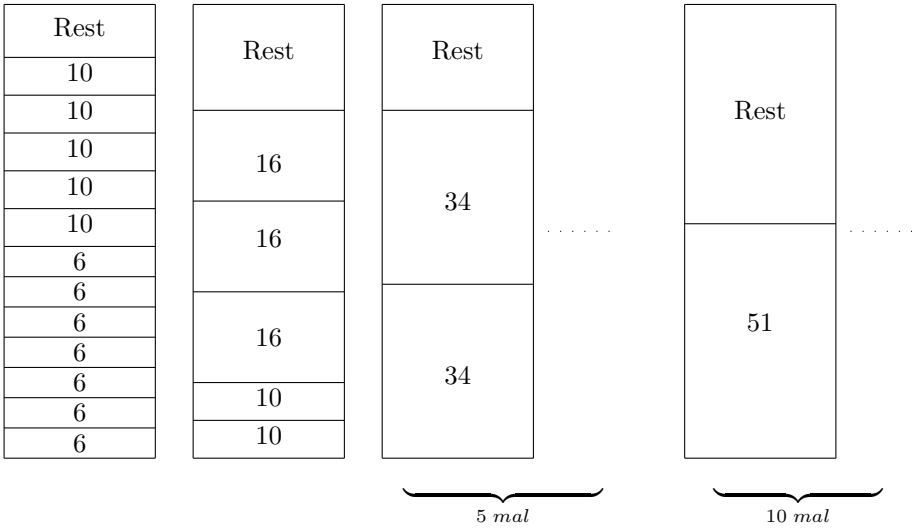
Mit $\mathrm{OPT}(I) \geq \sum_{a_i \in M} s(a_i)$ folgt die Behauptung.

**Beispiel 7.7.** Sei $B := 101$ „Größe“ der Mengen (statt 1) und $n = 37$.

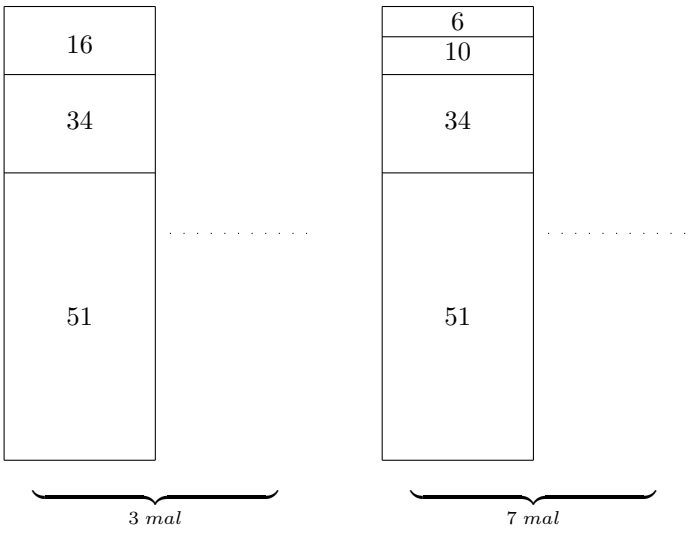
$$s(a_i) := \begin{cases} 6 & 1 \leq i \leq 7 \\ 10 & 8 \leq i \leq 14 \\ 16 & 15 \leq i \leq 17 \\ 34 & 18 \leq i \leq 27 \\ 51 & 28 \leq i \leq 37 \end{cases}$$

Dann ist $\mathrm{FF}(I) = 17$ und $\mathrm{OPT}(I) = 10$, d.h. $R_{FF}(I) = \frac{17}{10}$.

FF-Lösung:




Optimallösung:




■

**Satz 7.8.** *Für jedes Beispiel $I$ von* Bin Packing *gilt:*

$$FF(I) < \frac{17}{10} \cdot OPT(I) + 1.$$

*Beweis (Schematisch).* Definiere eine Funktion $\omega : [0,1] \longrightarrow [0,1]$, die man benutzt um das Ver-

hältnis von FF($I$) zu OPT($I$) abzuschätzen, indem man

$$\text{FF}(I) \quad \text{zu} \quad \omega(I) := \sum_{i=1}^{n} \omega(a_i)$$

beziehungsweise

$$\text{OPT}(I) \quad \text{zu} \quad \omega(I)$$

in Relation setzen. Sei $\omega(a)$ wie folgt definiert:

$$\omega(a) := \begin{cases} \frac{6}{5} \cdot a & \text{für } 0 \leq a < \frac{1}{6} \\ \frac{9}{5} \cdot a - \frac{1}{10} & \text{für } \frac{1}{6} \leq a < \frac{1}{3} \\ \frac{6}{5} \cdot a + \frac{1}{10} & \text{für } \frac{1}{3} \leq a \leq \frac{1}{2} \\ 1 & \text{für } \frac{1}{2} < a \leq 1 \end{cases}$$

Abkürzend benutzt man im Folgenden $\omega(a_i)$ für $\omega(s(a_i))$. Die Funktion $\omega$ erfüllt:

- Falls für $A \subseteq M$ gilt: $\sum_{a_i \in A} s(a_i) \leq 1$, dann gilt:

  $$\omega(A) := \sum_{a_i \in A} \omega(a_i) \leq \frac{17}{10}$$

  Daraus lässt sich herleiten, dass

  $$\sum_{a_i \in M} \omega(a_i) \leq \frac{17}{10} \cdot \text{OPT}(I),$$

  d.h. OPT($I$) kann man zu $\omega(I)$ in Relation setzen.

- Entsprechend verfährt man für FF($I$) und erhält:

  $$\sum_{a_i \in M} \omega(a_i) > FF(I) - 1.$$

□

**Bemerkung 7.9.** *Der Summand* 1 *ist bei dieser Abschätzung sicher vernachlässigbar. Definiere deshalb* asymptotische Gütegarantie $\mathcal{R}_{\mathcal{A}}^{\infty}$*:*

$$\mathcal{R}_{\mathcal{A}}^{\infty} := \inf \left\{ r \geq 1 : \begin{array}{l} \textit{Es gibt ein } N > 0, \textit{ so dass } R_{\mathcal{A}}(I) \leq r \\ \textit{für alle } I \textit{ mit } OPT(I) \geq N \end{array} \right\}$$

*Dann ist* $\mathcal{R}_{FF}^{\infty} = \frac{17}{10}$.

*Zu weiteren Approximationsalgorithmen für* Bin Packing *gibt es noch bessere asymptotische Gütegarantien als* $\frac{17}{10}$*. Als Beispiel sei hier* First Fit Decreasing *genannt mit einer asymptotischen Gütegarantie von* $\frac{11}{9}$.

## 7.2 Approximationsschemata

Kann es für $\mathcal{NP}$-schwere Optimierungsprobleme noch bessere Approximierbarkeitsresultate geben als Approximationsalgorithmen mit relativer Gütegarantie K, wobei K konstant ist? Die Antwort auf diese Frage wird Gegenstand dieses Abschnitts sein.

**Definition 7.10.** *Ein* (polynomiales) Approximationsschema (PAS) *für ein Optimierungsproblem* $\Pi$ *ist eine Familie von Algorithmen* $\{\mathcal{A}_\epsilon : \epsilon > 0\}$*, so dass* $\mathcal{A}_\epsilon$ *ein* $\epsilon$*-approxmierender Algorithmus ist, d.h.* $R_{\mathcal{A}_\epsilon} \leq 1 + \epsilon$ *für alle* $\epsilon > 0$*. Dabei bedeutet polynomial wie üblich polynomial in der Größe des Inputs I. Ein Approximationsschema* $\{\mathcal{A}_\epsilon : \epsilon > 0\}$ *heißt* vollpolynomial (FPAS)*, falls seine Laufzeit zudem polynomial in* $\frac{1}{\epsilon}$ *ist.*

Zunächst zeigen wir, dass für $\mathcal{NP}$-schwere Optimierungsprobleme ein FPAS in gewissem Sinne das Beste ist, was wir erwarten können. Man kann beweisen, dass folgender Satz gilt:

**Satz 7.11.** *Falls* $\mathcal{P} \neq \mathcal{NP}$ *und* $\Pi$ *ein* $\mathcal{NP}$*-schweres Optimierungsproblem ist, so gibt es kein PAS* $\{\mathcal{A}_\epsilon : \epsilon > 0\}$ *für* $\Pi$*, bei dem die Laufzeit von* $\mathcal{A}_\epsilon$ *zudem polynomial in* $\log \frac{1}{\epsilon}$ *(Kodierungslänge von* $\frac{1}{\epsilon}$*) ist.*

### 7.2.1 Ein PAS für Multiprocessor Scheduling

**Problem (**Multiprozessor Scheduling**).** *Gegeben seien n Jobs* $J_1, \ldots, J_n$ *mit Bearbeitungsdauer* $p_1, \ldots, p_n$ *und m identische Maschinen. Ohne Einschränkung sei* $m < n$*.*

*Gesucht ist nun ein Schedule mit minimalem* Makespan*, d.h. eine Zuordnung der n Jobs auf die m Maschinen, bei der zu keinem Zeitpunkt zwei Jobs auf einer Maschine liegen und der Zeitpunkt, zu dem alle Jobs abgearbeitet sind, minimal ist.*

$$\text{Makespan} := \max_{1 \leq j \leq m} \left( \sum_{J_i \text{ auf Maschine } j} p_i \right)$$

Multiprozessor Scheduling *ist* $\mathcal{NP}$*-vollständig.*

Zunächst wollen wir das einfache Verfahren List betrachten, das die Gütegarantie $2 - \frac{1}{m}$ besitzt. Betrachte $n$ Jobs $J_1, \ldots, J_n$ in beliebiger Reihenfolge als Liste angeordnet. Dieses Verfahren hat

**Algorithmus 47** : List Scheduling

**Eingabe** : $n$ Jobs $J_1, \ldots, J_n$ und $m$ Maschinen
**Ausgabe** : Zeiteffiziente Zuweisung der Jobs auf die Maschinen
1 Lege die ersten $m$ Jobs auf die $m$ Maschinen
2 Sobald eine Maschine frei ist, ordne ihr den nächsten der restlichen $n - m$ Jobs zu

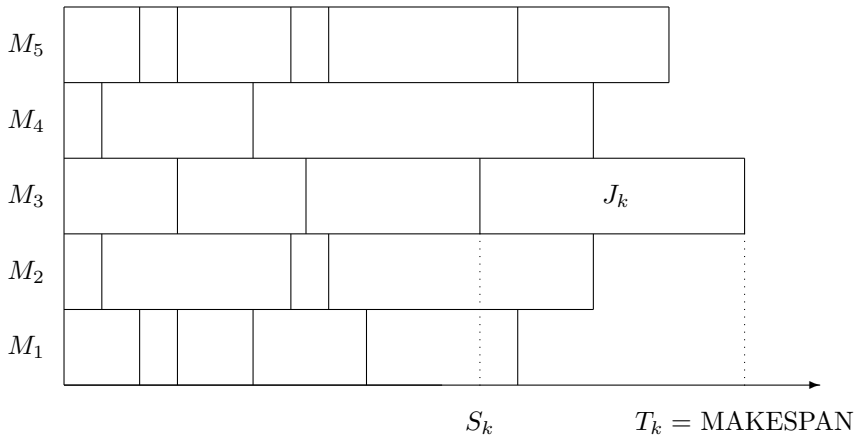
eine Laufzeit von $\mathcal{O}$(n). Es kann auch angewendet werden, ohne dass alle Jobs zu Beginn bekannt sind. Ein Beispiel hierfür sind Online-Szenarien.

**Satz 7.12.** *Für* List $\mathcal{A}$ *gilt:*

$$\mathcal{R}_\mathcal{A} = 2 - \frac{1}{m}$$

*Beweis.* Bezeichne $S_i$ die Startzeit von Job $J_i$ und $T_i$ die Abschlusszeit von Job $J_i$ im durch $\mathcal{A}$ konstruierten Schedule. Falls $J_k$ der zuletzt beendete Job ist, so ist $T_k = \text{Makespan}_\mathcal{A}$. Dann kann zu keinem Zeitpunkt vor $S_k$ irgendeine Maschine untätig (idle) sein, d.h.

$$S_k \leq \frac{1}{m} \cdot \sum_{j \neq k} p_j$$




Sei $T_{\text{OPT}}$ der optimale MAKESPAN. Für $T_{\text{OPT}}$ gilt:

$$\begin{aligned} T_{\text{OPT}} &\geq p_k \quad \text{da } J_k \text{ ausgeführt werden muss, und außerdem} \\ T_{\text{OPT}} &\geq \frac{1}{m} \cdot \sum_{j=1}^{m} p_j \end{aligned}$$

da diese untere Schranke die bestmögliche Auslastung der Maschinen repräsentiert. Andererseits gilt:

$$\begin{aligned} T_k &= S_k + p_k \\ &\leq \frac{1}{m} \cdot \sum_{j \neq k} p_j + p_k \\ &= \frac{1}{m} \cdot \sum_{j=1}^{m} p_j + \left(1 - \frac{1}{m}\right) \cdot p_k \\ &\leq T_{\text{OPT}} + \left(1 - \frac{1}{m}\right) \cdot T_{\text{OPT}} \\ &= \left(2 - \frac{1}{m}\right) \cdot T_{\text{OPT}} \end{aligned}$$

□

Basierend auf LIST kann man nun ein PAS für MULTIPROCESSOR SCHEDULING angeben:

**Algorithmus 48** : Algorithmus $\mathcal{A}_\ell$ für MULTIPROCESSOR SCHEDULING

**Eingabe** : $n$ Jobs $J_1, \ldots, J_n$ und konstantes $\ell$ $(1 \leq \ell \leq n)$
**Ausgabe** : Zeiteffiziente Zuteilung der Jobs auf die Maschinen

1 Bestimme zunächst einen optimalen Schedule für $J_1, \ldots, J_\ell$. Dabei sollen $J_1, \ldots, J_\ell, J_{\ell+1}$ die $\ell + 1$ längsten Jobs sein mit den Bearbeitungszeiten $p_1 \geq p_2 \geq \cdots \geq p_\ell \geq p_{\ell+1}$.
2 Ausgehend von diesem partiellen Schedule ordne den restlichen Jobs $J_{\ell+1}, \ldots, J_n$ Maschinen entsprechend LIST zu.

$\mathcal{A}_\ell$ kann in polynomialer Laufzeit realisiert werden, da $\ell$ konstant angenommen wird in $\mathcal{O}(m^\ell + n)$.

**Satz 7.13.** *Für $\mathcal{A}_\ell$ und $1 \leq \ell \leq n$ konstant, gilt:*

$$R_{\mathcal{A}_\ell} \leq 1 + \frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{\ell}{m} \right\rfloor}$$

*Beweis.* Es bezeichne $T$ die Abschlusszeit des Schedules für $J_1, \ldots, J_\ell$ nach Schritt 1. Falls die Abschlusszeit des Schedules für $J_1, \ldots, J_n$ auch $T$ ist, so ist $T = T_{\text{OPT}}$, d.h. $\mathcal{A}_\ell$ ist optimal. Sei also MAKESPAN $> T$ und $J_i$ mit $i > \ell$ der zuletzt beendete Job, d.h. $\text{MAKESPAN}_{\mathcal{A}_\ell} = T_i$, wobei $T_i$ die Abschlusszeit von $J_i$ ist. Im Zeitintervall $[0, T_i - p_i]$ muss dann jede Maschine belegt sein, da sonst $J_i$ schon früher begonnen worden wäre. Es folgt:

$$\sum_{j=1}^{n} p_j \geq m \cdot (T_i - p_i) + p_i = m \cdot T_i - (m-1) \cdot p_i.$$

Da $J_1, \ldots, J_\ell, J_{\ell+1}$ die $\ell + 1$ längsten Jobs sind, ist $p_i \leq p_{\ell+1}$, d.h. es gilt:

$$\begin{aligned} \sum_{j=1}^{n} p_j &\geq m \cdot (T_i - p_i) + p_i && (7.1)\\ &= m \cdot T_i - (m-1) \cdot p_i && (7.2)\\ &\geq m \cdot T_i - (m-1) \cdot p_{\ell+1}. && (7.3) \end{aligned}$$

Da zudem

$$T_{\text{OPT}} \geq \frac{1}{m} \cdot \sum_{j=1}^{n} p_j \geq \frac{1}{m} \cdot (m \cdot T_i - (m-1) \cdot p_{\ell+1})$$

ist, gilt mit Gleichung (7.3):

$$T_i \leq T_{\text{OPT}} + \frac{m-1}{m} \cdot p_{\ell+1} \ . \tag{7.4}$$

Andererseits gilt:

$$\begin{aligned} T_{\text{OPT}} &\geq p_{\ell+1} \cdot \left(1 + \left\lfloor \frac{\ell}{m} \right\rfloor\right), && (7.5)\\ & && (7.6) \end{aligned}$$

denn in einem optimalen Schedule muss mindestens eine Maschine mindestens $1 + \lfloor \frac{\ell}{m} \rfloor$ der Jobs $J_1, \ldots, J_\ell, J_{\ell+1}$ bearbeiten und außerdem ist

$$p_j \geq p_{\ell+1} \qquad \text{für } 1 \leq j \leq \ell. \tag{7.7}$$

Also ist wegen Gleichung (7.5)

$$p_{\ell+1} \leq \frac{T_{\text{OPT}}}{1 + \lfloor \frac{\ell}{m} \rfloor}$$

und damit folgt dann mit Gleichung (7.4):

$$\begin{aligned} R_{\mathcal{A}_\ell} = \frac{T_i}{T_{\text{OPT}}} &\leq 1 + \frac{1}{T_{\text{OPT}}} \cdot \left(1 - \frac{1}{m}\right) \cdot p_{l+1}\\ &\leq 1 + \left(1 - \frac{1}{m}\right) \cdot \frac{1}{1 + \lfloor \frac{\ell}{m} \rfloor} \end{aligned}$$

□

Aus $\mathcal{A}_\ell$ kann nun ein PAS abgeleitet werden. Zu $\epsilon > 0$ sei $\mathcal{A}_\epsilon$ ein Algorithmus $\mathcal{A}_\ell$ mit $\ell$ so gewählt, dass $R_{\mathcal{A}_\ell} \leq 1 + \epsilon$ ist, also

$$\frac{1 - \frac{1}{m}}{1 + \frac{\ell}{\lfloor m \rfloor}} \leq \epsilon \quad \text{ist.}$$

**Folgerung 7.14.** *Für* MULTIPROCESSOR SCHEDULING *mit konstanter Maschinenanzahl existiert ein PAS.*

**Bemerkung 7.15.** *Der Algorithmus 48 liefert* kein *FPAS* $\{\mathcal{A}_\ell : \ell > 0\}$*, da die Laufzeit von* $\mathcal{A}_\ell$ *in* $\mathcal{O}(m^\ell + n)$ *nicht polynomial in* $\frac{1}{\epsilon}$ *ist.*

## 7.3 Asymptotische PAS für Bin Packing

**Definition 7.16.** *Ein* asymptotisches PAS *- im Folgenden mit* APAS *abgekürzt - ist eine Familie von Algorithmen* $\{\mathcal{A}_\epsilon : \epsilon > 0\}$*, so dass* $\mathcal{A}_\epsilon$ *ein asymptotisch* $\epsilon$*-approximativer Algorithmus für jedes* $\epsilon > 0$ *ist, d.h.*

$$\mathcal{R}^{\infty}_{\mathcal{A}_\epsilon} \leq 1 + \epsilon.$$

*Entsprechend nennt man* $\{\mathcal{A}_\epsilon : \epsilon > 0\}$ *ein* asymptotisch vollpolynomiales PAS (AFPAS)*, wenn* $\mathcal{A}_\epsilon$ *zudem polynomial in* $\frac{1}{\epsilon}$ *ist.*

### 7.3.1 Ein APAS für Bin Packing

Ziel dieses Abschnittes ist es ein APAS $\{\mathcal{A}_\epsilon : \epsilon > 0\}$ mit

$$\mathcal{A}_\epsilon(I) \leq (1 + \epsilon) \cdot \mathrm{OPT}(I) + 1$$

für alle Instanzen $I$ mit Laufzeit $\mathcal{O}(n)$ zu entwickeln. Dabei sei $n$ die Anzahl der zu packenden Elemente, die exponentiell in $\frac{1}{\epsilon}$ ist. Eine Instanz $I$ für Bin Packing bestehe also aus den Elementen $\{1, \ldots, n\}$ mit den Größen $s_1, \ldots, s_n$, wobei $0 < s_i \leq 1$. Um das Problem zu lösen, benutzen wir folgende Techniken:

1. Restriktion von Bin Packing
2. Entfernen kleiner Elemente
3. Lineares Gruppieren

**Bemerkung 7.17.** *Es gilt offensichtlich für jede Instanz* $I$ *von* Bin Packing*:*

$$\begin{aligned} SIZE(I) := \sum_{i=1}^{n} s_i \quad &\leq \quad OPT(I) \quad \leq \quad n \quad \textit{und} \\ OPT(I) \quad &\leq \quad 2 \cdot SIZE(I) + 1 \end{aligned}$$

*da* Next Fit *immer eine solche Lösung findet. Ohne Einschränkung sei* $s_1 \geq \cdots \geq s_n \geq 0$.

Restricted Bin Packing **RBP**$[\delta, m]$

Gegeben seien Größen $V = \{v_1, \ldots, v_m\}$ mit $m < n$ derart, dass für diese gilt: $1 \geq v_1 > v_2 > \cdots > v_m \geq \delta > 0$. Eine Instanz von RBP$[\delta, m]$ besteht aus den Elementen $\{1, \ldots, n\}$, deren Größen $s_i$ alle aus $V$ sind, wobei jeweils $n_j$ Elemente der Größe $v_j$ vorkommen, d.h. $n = \sum_{j=1}^{m} n_j$.

Gesucht ist eine Partition von $\{1, \ldots, n\}$ mit minimaler Anzahl von Mengen $B_\ell$, so dass für alle $B_\ell$ gilt:

$$\sum_{j \in B_\ell} s_j \leq 1$$

Betrachte nun eine Lösung für eine Instanz $I$ von RBP$[\delta, m]$. Ein „BIN“ $B$ der Lösung ist dann charakterisiert durch ein $m$-Tupel $(b_1, \ldots, b_m)$, wobei $0 \leq b_j \leq n_j$ die Bedeutung „in $B$ sind $b_j$ Elemente der Größe $v_j$“ für $1 \leq j \leq m$ hat. Ein $m$-Tupel $T_t = (T_{t1}, \ldots, T_{tm})$ heißt *BIN-TYP*, falls $T_{tj} \in \mathbb{N}_0$ und

$$\sum_{j=1}^{m} T_{tj} \cdot v_j \leq 1 \ .$$

Für eine feste Menge $V = \{v_1, \ldots, v_m\}$ und $0 < \delta < 1$ bezeichnet $q$ die Anzahl möglicher, verschiedener *BIN-TYPEN*. Dann kann eine obere Schranke für $q$ angegeben werden, die nur von $\delta$ und $m$ abhängt, bezeichnet mit $q(\delta, m)$.

**Lemma 7.18.** *Sei* $k = \lfloor \frac{1}{\delta} \rfloor$. *Dann gilt:*

$$q(\delta, m) \leq \binom{m+k}{k}$$

*Beweis.* Ein *BIN-TYP* $(T_{t1}, \ldots, T_{tm})$ hat die Eigenschaft, dass

$$\sum_{j=1}^{m} T_{tj} \cdot v_j \leq 1 \text{ und } T_{tj} \geq 0 \text{ für } 1 \leq j \leq m.$$

Da $v_j \geq \delta$ für alle $j$, folgt

$$\sum_{j=1}^{m} T_{tj} \leq k \ .$$

Jeder *BIN-TYP* entspricht einer Möglichkeit (geordnet) $m$ nicht-negative ganze Zahlen zu wählen, die sich zu höchstens $k$ aufsummieren beziehungsweise $m+1$ nicht-negative ganze Zahlen zu wählen, die sich genau zu $k$ aufsummieren lassen. Die Anzahl dieser Möglichkeiten ist also die obere Schranke für $q(\delta, m)$. Dass diese Anzahl gerade

$$\binom{m+k}{k}$$

ist, lässt sich per Induktion mit einem einfachen Abzählargument beweisen. □

Eine Lösung einer Instanz $I$ von RBP$[\delta, m]$ kann also allein dadurch charakterisiert werden, wie viele BINs eines jeden der $q(\delta, m) = q$ *BIN-TYPEN* vorkommen, d.h. durch ein $q$-Tupel $X = (x_1, \ldots, x_q)$, wobei $x_t$ die Anzahl der BINs vom *BIN-TYP* $T_t$ angibt für $1 \leq t \leq q$.

Beachte, dass nicht alle *BIN-TYPEN* zu einer zulässigen Lösung gehören, da die Zulässigkeit für alle $j \in \{1, \ldots, m\}$ erfordert:

$$\sum_{t=1}^{q} x_t \cdot T_{tj} \leq n_j$$

**Beispiel 7.19.** Gegeben sei folgende Instanz $I$ mit $n = 20$, $m = 5$ und $\delta = \frac{1}{8}$. Ferner sei

| | |
|---|---|
| $v_1 = 1$ | $n_1 = 3$ |
| $v_2 = \frac{3}{4}$ | $n_2 = 3$ |
| $v_3 = \frac{1}{2}$ | $n_3 = 6$ |
| $v_4 = \frac{1}{4}$ | $n_4 = 3$ |
| $v_5 = \frac{1}{8}$ | $n_5 = 5$ |

Außerdem hat man

$$\begin{array}{lcllcl} T_1 & = & (1,0,0,0,0), & & T_2 & = & (0,1,0,0,0), \\ \ldots, & & (0,1,0,1,0), & \ldots, & T_t & = & (0,0,1,0,4), \\ \ldots, & & (0,0,0,2,4), & \ldots & & & \end{array}$$

als BIN-Typen. Es gibt maximal

$$\binom{5+8}{8} \geq q \text{ BIN-Typen.}$$

Eine Lösung ist X $= (x_1, \ldots, x_q)$, beispielsweise $(3, 3, \ldots, 0, \ldots, 1, \ldots)$. Ein $q$-Tupel X, bei dem zum Beispiel $T_1$ mindestens viermal oder $T_t$ mindestens zweimal vorkommt, ist keine zulässige Lösung. ■

**Lineares Programm zu RBP$[\delta, m]$**

Sei $A$ eine $q \times m$-Matrix, deren $t$-te Zeile dem $m$-Tupel $T_t$ entspricht und sei N $:= (n_1, \ldots, n_m)$. Dann ist

$$\forall j \in \{1, \ldots, m\} : \sum_{t=1}^{q} x_t \cdot T_{tj} = n_j \qquad \text{äquivalent zu} \qquad X \cdot A = N \ .$$

Die Anzahl der BINs in einer Lösung X ist einfach

$$\sum_{t=1}^{q} x_t = (1, \ldots, 1) \cdot X^T \ .$$

**Folgerung 7.20.** *Eine optimale Lösung von RBP$[\delta, m]$ entspricht einer ganzzahligen Lösung des folgenden „Integer linear Program“ (ILP(I)):*
*Minimiere*

$$1^T \cdot X^T$$

*unter der Bedingung $x_i \geq 0$ für $1 \leq i \leq q$ und*

$$X \cdot A = N.$$

Die Anzahl $q$ der Zeilen von $A$ ist exponentiell in $\delta$ und $m$. Wenn man jedoch voraussetzt, dass $\delta$ und $m$ konstant sind, so hängt die Größe des ILP$(I)$ nur von $n$ ab und das ILP$(I)$ kann in einer Zeit, die linear in $n$ ist, aufgestellt werden. Wie in Kapitel 6 erwähnt, ist Integer Linear Programming $\mathcal{NP}$-vollständig. Ein ILP, bei dem die Anzahl der Variablen konstant ist, kann jedoch in einer Laufzeit, die linear in der Anzahl der Nebenbedingungen ist, gelöst werden. Dieses Ergebnis stammt von H. W. Lenstra und ist aus dem Jahr 1983. Die Anzahl der Variablen ist hier $q$ und nur abhängig von $\delta$ und $m$.

**Satz 7.21.** *Eine Instanz $I$ von RBP$[\delta, m]$ kann in $\mathcal{O}(n + f(\delta, m))$ gelöst werden, wobei $f(\delta, m)$ eine Konstante ist, die nur von $\delta$ und $m$ abhängt.*

**Entfernen kleiner Elemente**

**Lemma 7.22.** *Sei $I$ eine Instanz für Bin Packing. Zu $\delta$ mit $0 < \delta \leq \frac{1}{2}$ sei eine Teillösung für $I$ gegeben, bei der alle Elemente der Größe $s_i > \delta$ in $\beta$ BINs gepackt werden können. Dann kann diese Teillösung erweitert werden zu einer Lösung von $I$, bei der die Anzahl der BINs höchstens*

$$\max\{\beta, (1 + 2 \cdot \delta) \cdot OPT(I) + 1\} \text{ ist.}$$

*Beweis.* Um das Lemma zu beweisen, benutzt man First Fit, um die Teillösung zu einer Lösung zu erweitern, wobei zunächst immer wieder die $\beta$ BINs der Teillösung behandelt werden. Wenn First Fit alle „kleineren“ Elemente in die ersten $\beta$ BINs packt, dann ist die Behauptung erfüllt. Bestehe also die Lösung aus $\beta' > \beta$ BINs. Wie bei der Abschätzung, dass allgemein

$$FF \leq 2 \cdot \mathrm{OPT} + 1$$

gilt, sind hier alle bis auf höchstens ein BIN mindestens zu $1 - \delta$ gefüllt. Also gilt nun:

$$SIZE(I) \quad \geq \quad (1 - \delta) \cdot (\beta' - 1)$$

Da

$$\begin{aligned} SIZE(I) &\leq \mathrm{OPT}(I) \qquad \text{folgt nun:} \\ \beta' &\leq \frac{1}{1-\delta} \cdot \mathrm{OPT}(I) + 1 \\ &\underbrace{\leq}_{0<\delta\leq\frac{1}{2}} \frac{1+\delta-2\cdot\delta^2}{1-\delta} \cdot \mathrm{OPT}(I) + 1 \\ &= (1+2\cdot\delta)\cdot \mathrm{OPT}(I) + 1 \ . \end{aligned}$$ □

**Lineares Gruppieren**

Eine Instanz $I$ von Bin Packing wird in eine Instanz von RBP$[\delta, m]$ überführt für ein geeignetes $\delta$ und $m$, ohne dass sich der Wert einer optimalen Lösung „zu sehr“ verändert. Zu einer Instanz $I$ von Bin Packing und zu $k$ sei $m := \lfloor \frac{n}{k} \rfloor$. Definiere ferner Gruppen $G_j$, $j \in \{1, \ldots, m+1\}$, von Elementen der Instanz $I$, so dass $G_1$ die $k$ größten Elemente, $G_2$ die $k$ nächstgrößten, usw. enthält, d.h.

$$\begin{aligned} G_j &:= \{(j-1)\cdot k + 1, \ldots, j\cdot k\} \qquad \text{für } j = 1, \ldots, m \text{ und} \\ G_{m+1} &:= \{m\cdot k + 1, \ldots, n\}, \end{aligned}$$

wobei die Größe von Element $i$ gerade $s_i$ sei, mit $1 \geq s_1 \geq s_2 \geq \cdots \geq s_n \geq \delta > 0$.

**Definition 7.23.** *Man definiert zu zwei Instanzen $I_1$ und $I_2$ von* Bin Packing *mit*

$$\begin{array}{ll} I_1 & \textit{enthält Elemente der Größe} \quad x_1 \geq x_2 \geq \cdots \geq x_n \\ I_2 & \textit{enthält Elemente der Größe} \quad y_1 \geq y_2 \geq \cdots \geq y_n \end{array}$$

*die Relation „$\geq$“ durch*

$$I_1 \geq I_2 \Longleftrightarrow x_i \geq y_i \qquad \textit{für alle } i \in \{1, \ldots, n\}.$$

*Man sagt dann „$I_1$ dominiert $I_2$“.*

**Folgerung 7.24.** *Falls $I_1 \geq I_2$, so gilt*

$$\begin{aligned} SIZE(I_1) &\geq SIZE(I_2) \qquad \textit{und} \\ OPT(I_1) &\geq OPT(I_2). \end{aligned}$$

*Nun gilt offensichtlich:*

$$G_1 \geq G_2 \geq G_3 \geq \cdots \geq G_m$$

Sei nun $v_j := s_{(j-1)\cdot k+1}$ Größe des größten Elementes in $G_j$. Definiere Gruppen $H_j$, $j \in \{1, \ldots, m\}$, bestehend aus jeweils $k$ Elementen der Größe $v_j$ und $H_{m+1}$ bestehend aus $|G_{m+1}|$ Elementen der Größe $v_{m+1}$. Dann gilt:

$$H_1 \geq G_1 \geq H_2 \geq G_2 \geq \cdots \geq H_m \geq G_m$$

Definiere nun zu einer Instanz I von Bin Packing zwei Instanzen

$$\begin{array}{lll} I_{LO} \text{ für „I Low“} & \text{aus} & H_2 \cup H_3 \cup \cdots \cup H_{m+1} \quad \text{(m Elemente!)} \\ I_{HI} \text{ für „I High“} & \text{aus} & H_1 \cup H_2 \cup \cdots \cup H_{m+1} \quad \text{(m+1 Elemente!)} \end{array}$$

Dann ist $I_{LO}$ eine Instanz von RBP$[\delta, m]$ und $I_{HI} \geq I$.

**Lemma 7.25.** *Es gilt:*

$$\begin{aligned} OPT(I_{LO}) \leq\ & OPT(I) \leq OPT(I_{HI}) \leq OPT(I_{LO}) + k \qquad \textit{und} \\ \mathrm{SIZE}(I_{LO}) \leq\ & \mathrm{SIZE}(I) \leq \mathrm{SIZE}(I_{HI}) \leq \mathrm{SIZE}(I_{LO}) + k \end{aligned}$$

*Beweis.* Betrachte Instanz $I_x$ bestehend aus $G_1 \cup G_2 \cup \cdots \cup G_{m-1} \cup X$, wobei $X \subseteq G_m$ mit $|X| = |G_{m+1}| = |H_{m+1}|$. Dann gilt $I_{LO} \leq I_x$ und daraus folgt:

$$\mathrm{OPT}(I_{LO}) \leq \mathrm{OPT}(I) \qquad \text{und} \qquad \mathrm{SIZE}(I_{LO}) \leq \mathrm{SIZE}(I)$$

$I_{HI}$ entsteht aus $I_{LO}$ durch Hinzufügen von $H_1$. Aus einer Lösung von $I_{LO}$ entsteht durch Hinzufügen von maximal $k$ zusätzlichen BINs eine Lösung von $I_{HI}$, also gilt

$$\begin{aligned} \mathrm{OPT}(I_{HI}) &\leq \mathrm{OPT}(I_{LO}) + k \quad \text{und trivialerweise gilt:} \\ \mathrm{SIZE}(I_{HI}) &\leq \mathrm{SIZE}(I_{LO}) + k \ . \end{aligned}$$

Die nun noch fehlende Ungleichung folgt aus $I \leq I_{HI}$. □

Mittels Sortieren kann aus einer beliebigen Instanz $I$ für Bin Packing $I_{LO}$ und $I_{HI}$ in $\mathcal{O}(n \log(n))$ konstruiert werden und aus einer optimalen Lösung für $I_{LO}$ eine Lösung für $I$ mit dem Wert $\mathrm{OPT}(I_{LO}) + k$.

### APAS $\{\mathcal{A}_\epsilon : \epsilon > 0\}$ für Bin Packing

Nach der geleisteten Vorarbeit kann man nun das vollständige APAS für Bin Packing formulieren:

**Algorithmus 49** : APAS für Bin Packing

**Eingabe** : Instanz $I$ mit $n$ Elementen der Größen $s_1 \geq \cdots \geq s_n$ und $\epsilon$
**Ausgabe** : Approximationslösung für Bin Packing

1 $\delta \leftarrow \frac{\epsilon}{2}$
2 Betrachte Instanz $J$ der Elemente aus $I$ mit Größe mindestens $\delta$. $J$ ist dann eine Instanz von RBP$[\delta, n']$.
3 $k \leftarrow \left\lceil \frac{\epsilon^2}{2} \cdot n' \right\rceil$
4 Berechne zu $J$ und $k$ Instanzen $J_{LO}$ von RBP$[\delta, m]$ und $J_{HI}$ bestehend aus $J_{LO}$ mit $H_1$, wobei $|H_1| = k$ ist und für $m := \lfloor \frac{n'}{k} \rfloor$ benutzt wird.
5 Berechne jetzt eine optimale Lösung von $J_{LO}$ durch optimales Lösen des ILP$(J_{LO})$.
6 Füge die $k$ Elemente aus $H_1$ in maximal $k$ zusätzliche BINs.
7 Berechne aus dieser Lösung für $J_{HI}$ Lösung für $J$ mit derselben Anzahl BINs.
8 Erweitere diese Lösung von $J$ mit First Fit zu einer Lösung von $I$.

**Satz 7.26.** *Für obigen Algorithmus gilt:*

$$\mathcal{A}_\epsilon(I) \leq (1+\epsilon) \cdot OPT(I) + 1 \qquad \textit{und}$$

*die Laufzeit von $\mathcal{A}_\epsilon$ ist in $\mathcal{O}(n \cdot \log(n) + c_\epsilon)$, wobei $c_\epsilon$ eine „Konstante“ ist, die von $\epsilon$ abhängt.*

*Beweis.* Die Laufzeit hängt nur vom Sortieren in Schritt 4 ab und von dem Lösen des ILP$(J_{LO})$ in Schritt 5. Die Laufzeit von ILP$(J_{LO})$ ist linear in $n$ und einer von $\epsilon$ abhängigen Konstanten. Die Lösung für $J$ benötigt maximal $\mathrm{OPT}(J_{LO}) + k$ viele BINs. Da alle Elemente in $J$ mindestens Größe $\delta = \frac{\epsilon}{2}$ haben, muss $\mathrm{SIZE}(J) \geq \epsilon \cdot \frac{n'}{2}$ sein. Also gilt:

$$k \leq \frac{\epsilon^2}{2} \cdot n' + 1 \leq \epsilon \cdot \mathrm{SIZE}(J) + 1 \leq \epsilon \cdot \mathrm{OPT}(J) + 1$$

Mit Lemma 7.25 folgt dann:

$$\mathrm{OPT}(J_{LO}) + k \leq \mathrm{OPT}(J) + \epsilon \cdot \mathrm{OPT}(J) + 1 \leq (1+\epsilon) \cdot \mathrm{OPT}(J) + 1$$

und somit folgt weiter:

$$\begin{aligned} \mathcal{A}_\epsilon(I) &\leq \max\{(1+\epsilon)\cdot \mathrm{OPT}(J)+1, (1+\epsilon)\cdot \mathrm{OPT}(I)+1\} \quad \text{und da} \\ \mathrm{OPT}(J) &\leq OPT(I) \quad \text{folgt weiter:} \\ \mathcal{A}_\epsilon(I) &\leq (1+\epsilon)\cdot \mathrm{OPT}(I)+1. \end{aligned}$$

□

**Bemerkung 7.27.** *Für die Laufzeit von $\mathcal{A}_\epsilon$ ist der Aufwand für das Lösen des ILP($J_{LO}$) in Schritt 5 entscheidend. Die Laufzeit ist exponentiell in $\frac{1}{\epsilon}$, da die Anzahl der Nebenbedingungen des ILP exponentiell in $\frac{1}{\epsilon}$ ist.*

### 7.3.2 AFPAS für Bin Packing

Aus dem konstruierten APAS kann ein AFPAS konstruiert werden, indem optimales Lösen des $ILP(J_{LO})$ ersetzt wird durch eine „nicht-ganzzahlige“ Lösung und darauf die Technik des „Rundens“ angewendet wird.

#### Runden nicht-ganzzahliger Lösungen

Man betrachtet wieder das Problem RBP$[\delta, m]$. Dieses Problem kann formuliert werden als:

$$\text{Minimiere} \quad 1^T \cdot X^T$$

unter den Bedingungen

- $x_i \geq 0$ für alle $1 \leq i \leq q$ und
- $X \cdot A = N$,

wobei $A$ eine $q \times m$-Matrix ist und $N$ ein $m$-Vektor. Man betrachtet nun die *Relaxierung* des ILPs, indem man die Ganzzahligkeitsbedingung für die $x_i$ weglässt. Man erhält dadurch ein *Lineares Programm (LP)* zu dem ILP. Bezeichne nun im Folgenden LIN($I$) den Wert einer (nicht notwendig ganzzahligen) optimalen Lösung des LP($I$). Die Interpretation dieses Resultats lautet wie folgt: Eine Lösung einer Bin Packing-Instanz $I$ von RBP$[\delta, m]$, die zu einer Lösung des LP($I$) korrespondiert, wäre eine Lösung, in der *Bruchteile von Elementen* in *Bruchteile von BINs* gepackt werden. Die Größe einer solchen optimalen Lösung ist gerade SIZE($I$). Allerdings würden die Bedingungen des LP keine beliebigen Bruchteile zulassen, sondern in jedem Bruchteil eines BINs müssten die Elemente denselben Bruchteil haben, d.h. anstatt ganzzahliger Anzahlen von *BIN-TYPEN* sind „gebrochenzahlige“ Anzahlen erlaubt.

#### Relaxierung des ILP

Aus der Theorie der linearen Programmierung übernehmen wir hier den Zusammenhang zwischen LPs und ILPs. Ohne Einschränkung sei dazu $rang(A) = m$ und die ersten $m$ Zeilen von $A$ bilden eine Basis. Dann heiße die Lösung $X^*$ mit $x_i = 0$ für $i > m$ *Basislösung*. Es gilt: Für jedes LP gibt es eine optimale Lösung, die eine Basislösung ist (siehe hierzu die Theorie der Linearen Programmierung).

Im Folgenden bezeichne *LIN(I)* den Wert einer nicht notwendig ganzzahligen, optimalen Lösung des LPs zu einer Instanz $I$.

**Lemma 7.28.** *Für alle Instanzen $I$ zu RBP$[\delta, m]$ gilt:*

$$\mathrm{SIZE}(I) \leq \mathrm{LIN}(I) \leq OPT(I) \leq \mathrm{LIN}(I) + \frac{m+1}{2}.$$

*Beweis.* Die ersten beiden Ungleichungen sind klar. Sei nun $Y$ eine Basislösung des LP($I$). Dann benutzt $Y$ höchstens $m$ verschiedene *BIN-TYPEN*. Würde man jede Komponente von $Y$ zum nächsten ganzzahligen Wert aufrunden, so würde der Wert der Lösung also um höchstens $m$ erhöht. Wir wollen hier allerdings eine schärfere Schranke (Summand $\frac{m+1}{2}$) beweisen. Dazu definiert man

$$\begin{array}{lllll} q\text{-Vektor} & W & \text{durch} & w_i := \lfloor Y_i \rfloor & \text{und} \\ & Z & \text{durch} & z_i := Y_i - w_i & \end{array}$$

mit $1 \leq i \leq q$, d.h. $W$ ist ganzzahliger und $Z$ ist gebrochenzahliger Anteil von $Y$. Sei $J$ Instanz von RBP$[\delta, m]$, die aus den Elementen von $I$ besteht, die nicht in der ganzzahligen Teillösung, die durch $W$ gegeben ist, in BINs gepackt werden. $Z$ induziert dann eine gebrochenzahlige Lösung für $J$, die höchstens $m$ *BIN-TYPEN* benutzt, die jeweils zu einem Bruchteil, der kleiner als 1 ist, benutzt werden. Also gilt:

$$\text{SIZE}(J) \leq \text{LIN}(J) \leq \sum_{i=1}^{q} z_i \leq m$$

Es gilt wiederum

$$\begin{array}{lll} \text{OPT}(J) & \leq & 2 \cdot \text{SIZE}(J) + 1 \quad \text{und offensichtlich ist wieder} \\ \text{OPT}(J) & \leq & m \ , \end{array}$$

da Aufrunden jedes positiven Eintrags in $Z$ auf 1 eine Lösung mit Wert $m$ liefert. Damit ergibt sich

$$\begin{array}{lll} \text{OPT}(J) & \leq & \min\{m, 2 \cdot \text{SIZE}(J) + 1\} \\ & \leq & \text{SIZE}(J) + \min\{m - \text{SIZE}(J), \text{SIZE}(J) + 1\} \\ & \leq & \text{SIZE}(J) + \dfrac{m+1}{2} \end{array}$$

Man muss jedoch OPT($I$) durch SIZE($I$) abschätzen und $m$ beschränken. Es gilt allerdings

$$\begin{array}{lllllll} \text{OPT}(I) & \leq & \text{OPT}(I-J) & + & \text{OPT}(J) & & \\ & \leq & \sum_{i=1}^{m} w_i & + & \text{SIZE}(J) & + & \frac{m+1}{2} \\ & \leq & \sum_{i=1}^{m} w_i & + & \text{LIN}(J) & + & \frac{m+1}{2} \\ & \leq & \sum_{i=1}^{m} w_i & + & \sum_{i=1}^{m} z_i & + & \frac{m+1}{2} \\ & = & \text{LIN}(I) & + & \frac{m+1}{2} & & \end{array}$$

□

**Bemerkung 7.29.** *Der Beweis zu obigem Lemma ist konstruktiv. Aus einer Lösung des LP($I$) kann man eine Lösung zu $I$ konstruieren mit obiger Schranke. Die Anzahl der Bedingungen in LP($I$) ist wieder exponentiell in $\frac{1}{\epsilon}$. Man kann aber folgenden Satz beweisen:*

**Satz 7.30.** *Es gibt einen vollpolynomialen Algorithmus $\mathcal{A}$ zur Lösung einer Instanz von RBP$[\delta, m]$ mit*

$$\mathcal{A}(I) \leq \text{LIN}(I) + \frac{m+1}{2} + 1.$$

*Dieses Verfahren stammt von Karmakar und Karp aus dem Jahre 1982.*

**Lemma 7.31.** *Mit „linearem Gruppieren“ kann zu einer Instanz $I$ von RBP$[\delta, m]$ eine Instanz $I_{LO}$ von RBP$[\delta, m]$ und eine Gruppe $H_1$ konstruiert werden, so dass für $I_{HI} = H_1 \cup I_{LO}$ gilt:*

$$\begin{array}{lll} \text{LIN}(I_{LO}) & \leq & \text{LIN}(I) \\ & \leq & \text{LIN}(I_{HI}) \\ & \leq & \text{LIN}(I_{LO}) + k \end{array}$$

Der Beweis zu diesem Lemma verläuft analog zum vorhergehenden Lemma.

Das Ergebnis lautet wie folgt. Man hat so ein AFPAS $\{\mathcal{A}_\epsilon : 1 \geq \epsilon > 0\}$ für eine Eingabe mit $n$ Elementen mit den Größen $\{s_1, \ldots, s_n\}$ erhalten. Ersetze nun im APAS Schritt 5 durch Schritt 5': Löse dann $J_{LO}$ entsprechend Satz 7.30.

**Satz 7.32.** *Es gilt die folgende Abschätzung:*

$$\mathcal{A}_\epsilon \leq (1+\epsilon) \cdot OPT(I) + \frac{1}{\epsilon^2} + 3$$

*Beweis.* Es gilt:

$$\mathrm{LIN}(J_{LO}) + 1 + \frac{m+1}{2} \leq \mathrm{OPT}(I) + \frac{1}{\epsilon^2} + 2$$

nach der Wahl von $m$ mit

$$m = \lfloor \frac{n'}{k} \rfloor = \frac{n'}{\frac{\epsilon^2}{2} \cdot n'} = \frac{2}{\epsilon^2}.$$

Andererseits ist

$$\begin{array}{rcl} OPT(J) & \geq & \mathrm{SIZE}(J) \geq n' \cdot \frac{\epsilon}{2} \quad \text{und damit ist} \\ k = \lceil \frac{n' \cdot \epsilon^2}{2} \rceil & \leq & 2 \cdot \mathrm{OPT}(J) + 1 \leq \epsilon \cdot \mathrm{OPT}(I) + 1. \end{array}$$

Nun ist aber

$$\begin{array}{rcl} \mathcal{A}_\epsilon & \leq & \mathrm{LIN}(J_0) + 1 + \frac{m+1}{2} + k \quad \text{und daraus folgt} \\ \mathcal{A}_\epsilon & \leq & (1+\epsilon) \cdot \mathrm{OPT}(I) + \frac{1}{\epsilon^2} + 3. \end{array}$$

□

# Kapitel 8

# Randomisierte Algorithmen

**Definition 8.1.** *Ein Algorithmus, der im Laufe seiner Ausführung gewisse Entscheidungen zufällig trifft, heißt* randomisierter Algorithmus.

**Beispiel 8.2.** Bei der randomisierten Variante von QUICKSORT wird das Element, nach dem in die Teilfolgen aufgeteilt wird, zufällig gewählt. ∎

Motivation für randomisierte Algorithmen:

- Für viele Probleme sind randomisierte Algorithmen schneller als deterministische Algorithmen.
- Typischerweise sind randomisierte Algorithmen einfacher zu beschreiben und zu implementieren als deterministische Algorithmen.

Man unterscheidet zwei Typen von randomisierten Algorithmen:

1. LAS VEGAS:

   Randomisierte Algorithmen, die immer ein korrektes Ergebnis liefern, gehören zu diesem Typ. In Abhängigkeit von den Wahlen, die zufällig getroffen werden, variiert die Laufzeit dieser Algorithmen. Man analysiert dann die Verteilung der Anzahlen der durchgeführten Rechnungsschritte.

2. MONTE CARLO:

   Randomisierte Algorithmen, die manchmal auch ein falsches Ergebnis liefern, fallen unter diese Kategorie von Algorithmen. Man untersucht hier die Wahrscheinlichkeit, mit der das Ergebnis falsch ist. Für Entscheidungsprobleme, d.h. deren mögliches Ergebnis JA/NEIN ist, gibt es zwei Arten von Monte Carlo-Algorithmen:

   (a) beidseitiger Fehler

   Ein Monte Carlo-Algorithmus hat einen beidseitigen Fehler, wenn für die beiden möglichen Antworten JA/NEIN die Wahrscheinlichkeit für eine falsche Antwort größer als Null ist.

   (b) einseitiger Fehler

   Ein Monte Carlo-Algorithmus hat einen einseitigen Fehler, wenn die Wahrscheinlichkeit, dass die Antwort falsch ist, in einem der beiden Fälle JA/NEIN gleich Null ist, d.h. zum Beispiel, wenn das Ergebnis „JA“ ausgegeben wird, ist dies immer richtig, während wenn „NEIN“ ausgegeben wird, ist dies nur mit einer bestimmten Wahrscheinlichkeit korrekt.

**Definition.** *1. Die Klasse $\mathcal{RP}$ (randomisiert polynomial) ist die Klasse der Entscheidungsprobleme $\Pi$, für die es einen polynomialen, randomisierten Algorithmus $A$ gibt, so dass für alle Instanzen $I$ von $\Pi$ gilt:*

$$\begin{cases} I \in Y_\Pi \longrightarrow \Pr[A(I) \text{ ist „}JA\text{“}] \geq \frac{1}{2} \\ I \notin Y_\Pi \longrightarrow \Pr[A(I) \text{ ist „}JA\text{“}] = 0 \end{cases}$$

*$Y_\Pi$ ist die Menge der sogenannten „JA-Beispiele“ von $\Pi$. Dabei entspricht $\Pr[A(I)$ ist „JA“$]$ der Wahrscheinlichkeit, dass die Antwort, die $A$ bei der Eingabe von $I$ gibt, „JA“ ist. Ein $\mathcal{RP}$-Algorithmus ist also ein einseitiger Monte Carlo-Algorithmus.*

*2. Die Klasse $\mathcal{PP}$ (probabilistic polynomial) ist die Klasse der Entscheidungsprobleme $\Pi$, für die es einen polynomialen Algorithmus $A$ gibt, so dass für alle Instanzen $I$ gilt:*

$$\begin{cases} I \in Y_\Pi \longrightarrow \Pr[A(I) \text{ ist „}JA\text{“}] > \frac{1}{2} \\ I \notin Y_\Pi \longrightarrow \Pr[A(I) \text{ ist „}JA\text{“}] < \frac{1}{2} \end{cases}$$

*Ein $\mathcal{PP}$-Algorithmus ist ein beidseitiger Monte Carlo-Algorithmus.*

*3. Die Klasse $\mathcal{BPP}$ (bounded error PP) ist die Klasse der Entscheidungsprobleme $\Pi$, für die es einen polynomialen Algorithmus $A$ gibt, so dass für alle Instanzen $I$ gilt:*

$$\begin{cases} I \in Y_\Pi \longrightarrow \Pr[A(I) \text{ ist „}JA\text{“}] \geq \frac{3}{4} \\ I \notin Y_\Pi \longrightarrow \Pr[A(I) \text{ ist „}JA\text{“}] \leq \frac{1}{4} \end{cases}$$

*Die probalistische Schranke kann zu $\frac{1}{2} + \frac{1}{p(n)}$ beziehungsweise $\frac{1}{2} - \frac{1}{p(n)}$ verschärft werden, wobei $p(n)$ ein Polynom in der Eingabegröße $n$ ist.*

## 8.1 Grundlagen der Wahrscheinlichkeitstheorie I

**Definition.** *1. Ein* Wahrscheinlichkeitsraum *ist ein Paar $(\Omega, \Pr)$, wobei $\Omega$ eine Menge und* Pr *eine Abbildung*

$$\Pr : \Omega \longrightarrow \mathbb{R}_0^+ \quad \text{ist mit} \quad \sum_{\omega \in \Omega} \Pr[\omega] = 1.$$

*2. Eine Teilmenge $A \subseteq \Omega$ heißt* Ereignis *und* Pr *wird erweitert auf Ereignisse durch*

$$\Pr[A] := \sum_{\omega \in A} \Pr[\omega].$$

*3. Die Elemente aus $\Omega$ heißen* Elementarereignisse.

*4. Falls $\Omega$ endlich ist und*

$$\Pr[\omega] = \frac{1}{|\Omega|} \qquad \text{für alle } \omega \in \Omega$$

*ist, so heißt* Pr Gleichverteilung über $\Omega$. *Im Folgenden bezeichne*

$$\Omega^+ := \{\omega \in \Omega : \Pr[\omega] > 0\}.$$

**Beispiel 8.5.**

1. „Fairer Würfel“

   Sei $\Omega := \{1, 2, 3, 4, 5, 6\}$ und $\Pr[d] = \frac{1}{6}$ für alle $d \in \Omega$. Dann ist

$$\Omega_{even} := \{2, 4, 6\} \text{ und damit folgt für } \Pr[\Omega_{even}] = 3 \cdot \frac{1}{6} = \frac{1}{2}\ .$$

2. „Zwei unabhängige Würfel“
   Sei $\Omega := \{1,2,3,4,5,6\} \times \{1,2,3,4,5,6\}$. Die Mächtigkeit von $\Omega$ ist dann $|\Omega| = 36$. Sei
   $\Omega_= := \{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$, dann gilt für diese:
   $$\begin{aligned} \Pr[\Omega_=] &= 6 \cdot \tfrac{1}{36} = \tfrac{1}{6} \text{ und für} \\ \Pr[\Omega_{\neq}] &= 1 - \tfrac{1}{6} = \tfrac{5}{6} \end{aligned}$$ ∎

**Definition.** *1. Seien $A_1$ und $A_2 \subseteq \Omega$ Ereignisse. Die* bedingte Wahrscheinlichkeit *von $A_1$ unter der Bedingung $A_2$ ist definiert als*

$$\Pr[A_1|A_2] := \frac{\Pr[A_1 \cap A_2]}{\Pr[A_2]},$$

*wobei $\Pr[A_2] > 0$.*

*2. Eine Menge von Ereignissen $\{A_i : i \in I\}$ heißt* unabhängig*, wenn für alle $S \subseteq I$ gilt:*

$$\Pr[\cap_{i \in S} A_i] = \Pi_{i \in S} \Pr[A_i]$$

Für Ereignisse $A_1$ und $A_2$ gilt:

$$\begin{aligned} \Pr[A_1 \cap A_2] &= \Pr[A_1|A_2] \cdot \Pr[A_2] \\ &= \Pr[A_2|A_1] \cdot \Pr[A_1] \end{aligned}$$

Per Induktion kann man zeigen, dass für Ereignisse $A_1, \ldots, A_\ell$ gilt:

$$\Pr\left[\bigcap_{i=1}^{\ell} A_i\right] = \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_1 \cap A_2] \cdot \ldots \cdot \Pr\left[A_\ell | \bigcap_{i=1}^{\ell-1} A_i\right]$$

## 8.2 Randomisierte MinCut-Algorithmen

### 8.2.1 Ein einfacher Monte Carlo-Algorithmus für MinCut

Man betrachtet hier folgendes MinCut-Problem: Man fasst $G = (V, E)$ mit $c : E \to \mathbb{N}$ auf als Multigraph, d.h. wenn $c(\{u, v\}) = \ell$, so gibt es im Multigraph $\ell$ Kanten, die $u$ und $v$ verbinden. Bezeichne nun also $G = (V, E)$ einen solchen Multigraphen. Gesucht ist nun eine Partition $V_1$ und $V_2$ von $V$, so dass

$$\text{cutsize}(V_1, V_2) := \left| \left\{ e \in E : \begin{array}{l} e \text{ verbindet Knoten } u \text{ und } v \text{ mit} \\ u \in V_1 \text{ und } v \in V_2 \text{ oder umgekehrt} \end{array} \right\} \right|$$

minimal ist.

**Algorithmus 50** : RANDOM MINCUT

**Eingabe** : Graph $G = (V, E)$ als Multigraph
**Ausgabe** : Schnitt in Form eines Graphen mit zwei Superknoten

```
1 Solange |V| > 2 tue
2     e ← zufällige Kante in E
3     Bilde neuen Graph G = (V, E), der entsteht, wenn die Endknoten von e verschmolzen
      werden und alle Kanten zwischen Endknoten von e entfernt werden
```

In jedem Schritt nimmt $|V|$ um 1 ab, d.h. nach $n-2$ Schritten endet das Verfahren mit 2 Knoten $v_1$ und $v_2$, die einen Schnitt $(V_1, V_2)$ des Ausgangsgraphen $G$ induzieren. Nun stellt sich die Frage, wie groß die Wahrscheinlichkeit ist, dass RANDOM MINCUT einen minimalen Schnitt liefert. Die Antwort hierauf liefert der nächste Satz:

**Satz 8.7.** *Die Wahrscheinlichkeit, dass* RANDOM MINCUT *einen bestimmten minimalen Schnitt (der möglicherweise auch der einzige ist) findet, mit der Bedingung, dass alle Kanten die gleiche Wahrscheinlichkeit haben gewählt zu werden, ist größer als* $\frac{2}{n^2}$*, wobei* $|V| = n$*.*

*Beweis.* Sei $(V_1, V_2)$ ein beliebiger, vorgegebener minimaler Schnitt von $G$ mit $k$ Kanten. Dann hat $G$ mindestens $k \cdot \frac{n}{2}$ Kanten, da alle Knoten in $G$ mindestens Grad $k$ haben (sonst gäbe es einen kleineren Schnitt). Man schätzt nun die Wahrscheinlichkeit, dass während der Durchführung von RANDOM MINCUT niemals eine Kante zwischen $V_1$ und $V_2$ gewählt wird, ab. Sei $A_i$ das Ereignis, dass im i-ten Schritt keine Kante aus $(V_1, V_2)$ gewählt wird $(1 \leq i \leq n-2)$. Dann ist

$$\Pr[A_1] \geq 1 - \frac{2}{n},$$

da die Wahrscheinlichkeit, dass im ersten Schritt gerade eine Kante aus $(V_1, V_2)$ gewählt wird, höchstens $k/\frac{k \cdot n}{2}$ ist. Nach dem ersten Schritt gibt es mindestens noch $k \cdot \frac{n-1}{2}$ Kanten. Entsprechend ist die Wahrscheinlichkeit, dass im zweiten Schritt eine Kante aus $(V_1, V_2)$ gewählt wird, nachdem im ersten Schritt $A_1$ eingetreten ist höchstens

$$\frac{k}{\frac{k \cdot (n-1)}{2}}, \quad \text{also} \quad \Pr[A_2 | A_1] \geq 1 - \frac{2}{n-1}.$$

Beim $i$-ten Schritt gibt es $n - i + 1$ Knoten und damit also mindestens $k \cdot \frac{(n-i+1)}{2}$ Kanten. Nun folgt:

$$\Pr\left[A_i | \bigcap_{j=1}^{i-1} A_j\right] \geq 1 - \frac{2}{n-i+1}$$

Die Wahrscheinlichkeit, dass in keinem der $n-2$ Schritte eine Kante aus $(V_1, V_2)$ gewählt wird, ist dann

$$\begin{aligned} \Pr\left[\bigcap_{i=1}^{n-2} A_i\right] &\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) \\ &= \frac{2}{n \cdot (n-1)} \\ &= \frac{1}{\binom{n}{2}}. \end{aligned}$$

□

**Folgerung 8.8.** *Wendet man* RANDOM MINCUT *nur* $n - \ell$ *Schritte lang an, d.h. man stoppt, wenn* $\ell$ *Knoten übrig sind, so ist die Wahrscheinlichkeit, dass bis dahin keine Kante eines bestimmten minimalen Schnitts* $(V_1, V_2)$ *gewählt wurde, mindestens*

$$\frac{\binom{\ell}{2}}{\binom{n}{2}}, \qquad \textit{d.h. in } \Omega\left(\left(\frac{\ell}{n}\right)^2\right).$$

Wenn man die Wahl einer zufälligen Kante in $O(n)$ realisieren kann, so hat Algorithmus 50 eine Laufzeit von $O(n^2)$. Diese Laufzeit ist deutlich besser als die des deterministischen MINCUT-Algorithmus (siehe Algorithmus 32). Wendet man RANDOM MINCUT $\frac{n^2}{2}$ mal unabhängig voneinander an, so ist die Wahrscheinlichkeit, dass ein bestimmter Schnitt nicht gefunden wurde, höchstens

$$\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}} < \frac{1}{e} \qquad \text{wobei } e \text{ die EULERsche Zahl ist.}$$

Dies ist wiederum schlechter als beim deterministischen MINCUT-Algorithmus (vgl. Kapitel 3).

### 8.2.2 Ein effizienterer randomisierter MinCut-Algorithmus

Um mit Random MinCut eine gute, d.h. geringe Fehlerwahrscheinlichkeit zu garantieren, muss man ihn „oft“ wiederholen. Um die Schranke $\frac{1}{e}$ zu garantieren benötigt man eine Laufzeit von $O(n^4)$. Diese Laufzeit kann mit folgender Idee verbessert werden:

Wende Random MinCut so viele Schritte an, bis der Graph mit $\frac{n}{\sqrt{2}}$ Knoten übrig ist und berechne darin rekursiv (random.) einen minimalen Schnitt. Dies wird zweimal ausgeführt und der kleinere der beiden Schnitte ausgewählt.

**Algorithmus 51** : Fast Random MinCut

**Eingabe** : Graph $G = (V, E)$ als Multigraph, $|V| = n$
**Ausgabe** : Schnitt

1 **Wenn** $n \leq 6$
2 berechne direkt deterministisch einen MinCut
3 **sonst**
4 $\ell \leftarrow \left\lceil \frac{n}{\sqrt{2}} \right\rceil$
5 $G_1 \leftarrow$ Random MinCut(bis $\ell$ Knoten übrig)
6 $G_2 \leftarrow$ Random MinCut(bis $\ell$ Knoten übrig)
7 $C_1 \leftarrow$ Fast Random MinCut($G_1$) (rekursiv)
8 $C_2 \leftarrow$ Fast Random MinCut($G_2$) (rekursiv)
9 Gib den kleineren der beiden Schnitte $C_1$ und $C_2$ aus

**Satz 8.9.** Fast Random MinCut *hat eine Laufzeit von* $\mathcal{O}(n^2 \cdot \log n)$.

*Beweis.* Die Laufzeit $T(n)$ ergibt sich aus der folgenden Rekursionsabschätzung:

$$T(n) = \underbrace{2 \cdot T\left(\left\lceil \frac{n}{\sqrt{2}} \right\rceil\right)}_{\text{Schritte 7 und 8}} + \underbrace{c \cdot n^2}_{\text{Schritte 5 und 6}}$$

Dabei ist $c$ eine Konstante. Es folgt unmittelbar, dass $T(n) \in \mathcal{O}(n^2 \cdot \log n)$ ist. □

**Satz 8.10.** *Die Wahrscheinlichkeit, dass* Fast Random MinCut *einen minimalen Schnitt findet, ist in* $\Omega\left(\frac{1}{\log n}\right)$.

*Beweis (Idee).* Der Beweis von Satz 8.10 ist lang und schwierig. Deshalb ist hier nur eine Beweisskizze angegeben. Angenommen die Größe eines minimalen Schnittes habe $k$ Kanten und angenommen es existiert ein Graph $G'$ mit $\ell$ Knoten, der durch Verschmelzung von Knoten aus $G$ entstanden ist und ebenfalls einen Schnitt mit $k$ Kanten habe. Jetzt betrachtet man einen Durchlauf für $G'$ von Fast Random MinCut. Das Ergebnis wird genau dann ein minimaler Schnitt von $G'$ sein (und damit für $G$), wenn die Rekursion für $G_1$ oder für $G_2$ einen Schnitt der Größe $k$ ausgibt. Die Wahrscheinlichkeit, dass bei der Berechnung von $G'$ keine Kante eines bestimmten Schnittes ausgewählt wurde, ist mindestens

$$\left\lceil \frac{\ell}{\sqrt{2}} \right\rceil \cdot \frac{\left\lceil \frac{\ell}{\sqrt{2}} \right\rceil - 1}{\ell \cdot (\ell - 1)} \geq \frac{1}{2} .$$

Bezeichne $P(\ell)$ die Wahrscheinlichkeit, dass Fast Random MinCut in einem Graph mit $\ell$ Knoten einen minimalen Schnitt findet, so folgt:

$$\begin{aligned} P(\ell) &\geq 1 - \left(1 - \frac{1}{2} \cdot P\left(\left\lceil \frac{\ell}{\sqrt{2}} \right\rceil\right)\right)^2 \\ &= P\left(\left\lceil \frac{\ell}{\sqrt{2}} \right\rceil\right) - \frac{1}{4} \cdot P\left(\frac{\ell}{\sqrt{2}}\right)^2 \end{aligned}$$

Setze nun $\ell = \sqrt{2^{k+1}}$, dann folgt

$$P\left(\sqrt{2^{k+1}}\right) \geq P\left(\left(\sqrt{2}\right)^k\right) - \frac{1}{4} \cdot P\left(\left(\sqrt{2}\right)^k\right)^2 ,$$

also wenn

$$s(k) \quad := \quad P\left(\left(\sqrt{2}\right)^k\right), \quad \text{so ist} \quad s(k+1) \geq s(k) - \frac{1}{4} \cdot s(k)^2 \ .$$

Wenn man nun

$$\begin{aligned} q(k) &:= \frac{4}{s(k)-1} \qquad \text{setzt, d.h.} \\ s(k) &= \frac{4}{q(k)+1} \qquad \text{, dann folgt:} \end{aligned}$$

$$\begin{aligned} s(k+1) = \frac{4}{q(k+1)+1} &\geq \frac{4}{q(k)+1} - \frac{4}{(q(k)+1)^2} \\ \implies \quad q(k)+1 &\geq q(k+1)+1 - \frac{q(k+1)+1}{q(k)+1} \\ &= (q(k+1)+1) \cdot \left(1 - \frac{1}{q(k)+1}\right) \\ \implies \quad q(k+1)+1 &\leq (q(k)+1) \cdot \left(\frac{1}{1-\frac{1}{q(k)+1}}\right) \\ q(k+1) &\leq q(k)+1-1+\frac{\frac{1}{q(k)+1}}{1-\frac{1}{q(k)+1}} \\ &= \frac{q(k)+\frac{1}{q(k)+1}}{1-\frac{1}{q(k)+1}} \\ &= q(k)+1+\frac{1}{q(k)} \ . \end{aligned}$$

Induktiv lässt sich nun zeigen, dass

$$q(k) < k + \sum_{i=1}^{k-1} \frac{1}{i} + 3 \quad \in \quad \Theta(k + \log k)$$

ist. Daraus folgt $s(k) \in \Omega\left(\frac{1}{k}\right)$ und $P(\ell) \in \Omega\left(\frac{1}{\log \ell}\right)$. □

## 8.3 Grundlagen der Wahrscheinlichkeitstheorie II

**Definition 8.11.** *Zu einem Wahrscheinlichkeitsraum* $(\Omega, \Pr)$ *heißt eine Funktion* $X$*, definiert als Abbildung*

$$X \colon \Omega \longrightarrow \mathbb{R}$$

Zufallsvariable.

Die Definition der Zufallsvariablen ermöglicht die Darstellung komplexer Ereignisse in kompakter Form. Man schreibt:

$$\begin{aligned} X = x \quad &\text{für} \quad \{\omega \in \Omega \mid X(\omega) = x\} \quad \text{und} \\ \Pr[X = x] \quad &\text{für} \quad \Pr[\{\omega \in \Omega \mid X(\omega) = x\}] \end{aligned}$$

**Definition 8.12.**

1. *Zwei Zufallsvariablen $X$ und $Y$ heißen* unabhängige *Zufallsvariablen, falls*
$$\Pr[X = x \ \wedge \ Y = y] = \Pr[X = x] \cdot \Pr[Y = y] \ .$$
2. *Der* Erwartungswert $E(X)$ *einer Zufallsvariablen $X$ ist definiert durch*
$$E(X) := \sum_{x \in X(\Omega^*)} x \cdot \Pr[X = x] \ ,$$
*wobei $\Omega^*$ die Menge aller Elementarereignisse $\Omega$ mit positiver Wahrscheinlichkeit ist.*

**Beispiel 8.13.** Zufallsvariablen für den fairen Würfel:

X : obenliegender Würfelseite wird entsprechende Punktzahl zugeordnet

X': obenliegender Würfelseite wird entsprechende Punktzahl der Rückseite zugeordnet

Für den Erwartungswert bei einem Würfel gilt:

$$E(X) = \sum_{i=1}^{6} i \cdot \frac{1}{6} = \frac{21}{6} = \frac{7}{2} = 3.5$$

■

Aus der Wahrscheinlichkeitstheorie sind ferner folgende Ergebnisse bekannt:

1. Für Zufallsvariablen $X$ und $Y$ und einen skalaren Faktor $c \in \mathbb{R}$ gilt:
$$E(c \cdot X) = c \cdot E(X) \quad \text{und}$$
$$E(X + Y) = E(X) + E(Y)$$
2. Falls $X$ und $Y$ unabhängige Zufallsvariablen sind, so gilt:
$$E(X \cdot Y) = E(X) \cdot E(Y)$$

## 8.4 Das Maximum Satisfiability Problem

**Problem (**Maximum Satisfiability Problem**).** *Gegeben ist eine Menge von $m$ Klauseln über einer Variablenmenge $V$ mit der Mächtigkeit $|V| = n$. Gesucht ist eine Wahrheitsbelegung, die eine maximale Anzahl von Klauseln erfüllt.*

**Bemerkung 8.14.** *Das* Maximum Satisfiability Problem*, auch bekannt als* Maximum Sat*, ist ein $\mathcal{NP}$-schweres Problem. Es ist sogar schon $\mathcal{NP}$-schwer, wenn man die Anzahl der Literale pro Klausel auf maximal 2 beschränkt (man spricht dann vom* Max-2-Sat*-Problem).*

**Beispiel 8.15.**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1. Klausel: | $X_1$ | $\vee$ | $\overline{X_2}$ | 2. Klausel: | $\overline{X_1}$ | $\vee$ | $\overline{X_2}$ |
| 3. Klausel: | $X_1$ | $\vee$ | $X_2$ | 4. Klausel: | $\overline{X_1}$ | $\vee$ | $X_3$ |
| 5. Klausel: | $X_2$ | $\vee$ | $\overline{X_3}$ | | | | |

Diese Klauseln sind nicht alle gleichzeitig erfüllbar, denn falls zum Beispiel $X_1 = falsch$, so ist $\overline{X_2}$ = wahr und aus der 3. Klausel würde folgen, dass auch $X_2$ = wahr sein müsste, also Widerspruch. Belegt man nun $X_1$ = wahr, so folgt $\overline{X_2}$ = wahr und $X_3$ = wahr, aber die 5. Klausel liefert dann falsch zurück. Eine maximale Anzahl von Klauseln mit wahr zu belegen, liefert $X_1$ = wahr, $X_2 = falsch$ und $X_3$ = wahr. Dann sind 4 von 5 Klauseln erfüllt. ■

### 8.4.1 Der Algorithmus Random Sat

Für jede Variable $x \in V$ setze $\omega(x) := \mathit{wahr}$ mit der Wahrscheinlichkeit $\frac{1}{2}$, wobei $V$ eine Menge von Variablen ist mit der Mächtigkeit $|V| = n$. Bezeichne $X_{RS}(I)$ die Zufallsvariable, die den Wert der Lösung von RANDOM SAT bei der Eingabe von $I$ angibt.

**Satz 8.16.** *Für eine Instanz $I$ von* MAX SAT *mit $m$ Klauseln, in der jede Klauseln mindestens $k$ Literale enthält, erfüllt der erwartete Wert der Lösung von* RANDOM SAT*:*

$$E(X_{RS}(I)) \geq \left(1 - \frac{1}{2^k}\right) \cdot m$$

*Beweis.* Die Wahrscheinlichkeit, dass eine Klausel mit $k$ Literalen nicht erfüllt wird, ist $\frac{1}{2^k}$. Entsprechend ist die Wahrscheinlichkeit, dass eine Klausel mit mindestens $k$ Literalen erfüllt wird mindestens $1 - \frac{1}{2^k}$. Damit ist der erwartete Beitrag einer Klausel zu $E(X_{RS}(I))$ mindestens $1 - \frac{1}{2^k}$, also folgt:

$$E(X_{RS}(I)) \geq (1 - \frac{1}{2^k}) \cdot m$$

□

**Korollar 8.17.** RANDOM SAT *ist 2-approximativ, d.h.*

$$\frac{OPT(I)}{E[X_{RS(I)}]} \leq 2$$

## 8.5 Das MaxCut-Problem

> **Der Literaturtip.** Die hier behandelten Ideen wurden im Jahre 1995 von Goemans und Williamson entdeckt ([10]).

**Problem.** *Gegeben ist ein Graph $G = (V, E)$ mit Gewichtsfunktion $c\colon E \longrightarrow \mathbb{N}$. Gesucht ist ein Schnitt $(S, V \setminus S)$ von $G$ mit maximalen Gewicht, d.h.*

$$c(S, V \setminus S) := \sum_{u,v \,\in\, E} c(\{u, v\}) \qquad \textit{soll maximal sein, wobei}$$

*$u \in S$ und $v \in V \setminus S$ ist.* MAXCUT *ist ein ähnliches Problem wie* MINCUT*, aber im Gegensatz zu diesem ist es $\mathcal{NP}$-schwer.*

### 8.5.1 Ein Randomisierter Algorithmus für MAXCUT basierend auf semidefiniter Programmierung

Sei $I$ eine Instanz für MAXCUT. Dann definiere dazu ein ganzzahliges quadratisches Programm $IQP(I)$ wie folgt. Zu $i$ und $j \in V := \{1, \ldots, n\}$ definiere

$$c_{ij} := \begin{cases} c(\{i, j\}) & \text{falls } \{i, j\} \in E \\ 0 & \text{sonst} \end{cases}$$

$(c_{ij})$ heißt *Gewichtsmatrix* zum Graphen $G$. Das IQP$(I)$ ist nun

$$\max \frac{1}{2} \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij} \cdot (1 - x_i \cdot x_j)$$

unter den Nebenbedingungen

$$x_i, x_j \;\in\; \{-1, 1\} \quad \text{und} \quad 1 \leq i, j \leq n \;.$$

$x_i = 1$ steht dann für $i \in S$ und $x_i = -1$ für $i \in V \setminus S$. Dann induziert die Belegung der $x_i$ und $x_j$ eine Partition $(S, V \setminus S)$ von $V$ mit

$$c(S, V \setminus S) = \frac{1}{2} \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij} \cdot (1 - x_i \cdot x_j) \ .$$

Denn falls $i, j \in S$ oder $i, j \in V \setminus S$, so gilt $x_i = x_j$ und damit folgt $(1 - x_i \cdot x_j) = 0$, andernfalls erhält man $\frac{1}{2} \cdot (1 - x_i \cdot x_j) = 1$. Eine optimale Lösung von IQP($I$) induziert also einen MaxCut zu $I$. Jede Variable $x_i$ kann auch als ein eindimensionaler Vektor der Norm 1 aufgefasst werden.

### 8.5.2 Relaxierung von IQP $(I)$

Sei $x^i$ ein normierter Vektor im zweidimensionalen Raum und sei $\mathrm{QP}^2(I)$ definiert als

$$\max \frac{1}{2} \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij} \cdot (1 - x^i \cdot x^j)$$

unter den Nebenbedingungen

$$x^i, x^j \in \mathbb{R}^2 \text{ mit Norm 1 für } 1 \leq i, j \leq n \ .$$

Damit gilt für das Produkt von $x^i$ und $x^j$:

$$x^i \cdot x^j = x_1^i \cdot x_1^j + x_2^i \cdot x_2^j$$

$\mathrm{QP}^2(I)$ ist tatsächlich eine Relaxierung von IQP($I$), denn jede Lösung $(x_1, \ldots, x_n)$ von IQP($I$) induziert eine Lösung $(x^1, \ldots, x^n)$ von $\mathrm{QP}^2(I)$ mittels $x^i = (x_i, 0)$.

#### Idee eines randomisierten Algorithmus zur Lösung des IQP

Berechne eine optimale Lösung $(\widetilde{x^1}, \ldots, \widetilde{x^n})$ von $\mathrm{QP}^2(I)$ und konstruiere daraus die Lösung $(S, V \setminus S)$ zu IQP($I$) mittels eines zufällig gewählten, zweidimensionalen, normierten Vektors $r$: In $S$ seien genau die Knoten $i \in V$ enthalten, für die der Vektor $\widetilde{x^i}$ oberhalb der zu $r$ senkrechten Linie $\ell$ liegt (vgl. Abbildung 8.1).

**Algorithmus 52** : Random MaxCut

**Eingabe** : Graph $G = (V, E)$ mit einer Gewichtsfunktion $c : E \longrightarrow \mathbb{N}$
**Ausgabe** : Ein Schnitt $(S, V \setminus S)$ in $G$

1 Stelle $\mathrm{QP}^2$ auf
2 Berechne die optimale Lösung $(\widetilde{x^1}, \ldots, \widetilde{x^n})$ des $\mathrm{QP}^2$
3 Wähle zufällig einen zweidimensionalen Vektor $r$ mit Norm 1
4 $S \leftarrow \{i \in V : \widetilde{x^i} \cdot r \geq 0\}$

**Satz 8.18.** *Sei $I$ eine Instanz für* MaxCut *und $C_{\mathrm{RMC}}(I)$ der Wert der Lösung, die* Random MaxCut *für $I$ berechnet. Wenn die Vektoren $r$ in Schritt 3 gleichverteilt angenommen werden, so gilt:*

$$E(C_{RMC}(I)) = \frac{1}{\pi} \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij} \cdot \arccos(\widetilde{x^i} \cdot \widetilde{x^j})$$

*Beweis.* Definiere $\mathrm{sgn} : \mathbb{R} \longrightarrow \{1, -1\}$ als

$$\mathrm{sgn}(x) := \begin{cases} 1 & \text{falls} \quad x \geq 0 \\ -1 & \text{sonst} \end{cases}$$

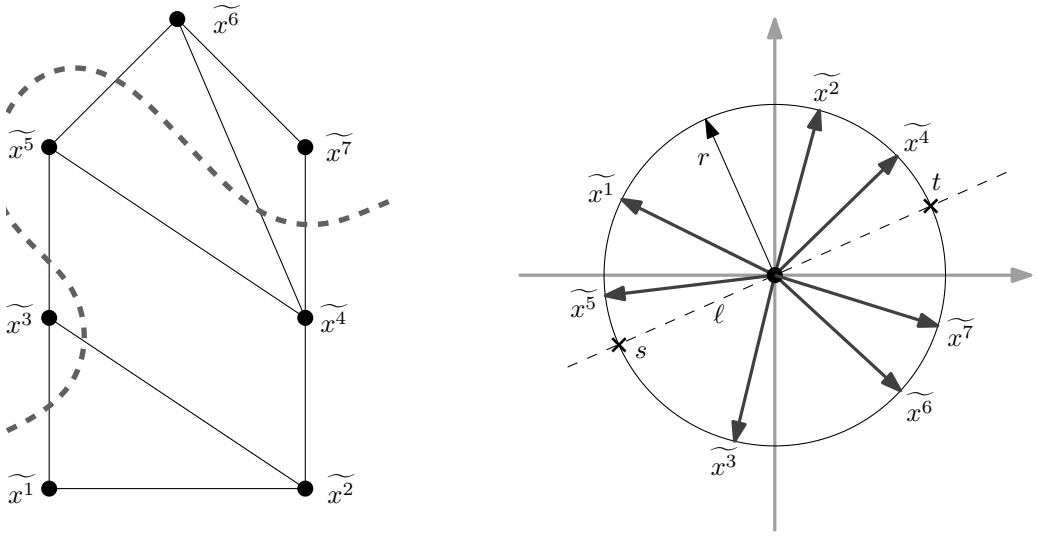



Abbildung 8.1: Eine Lösung von $\mathrm{QP}^2(I)$.

Offensichtlich gilt:

$$E(C_{\mathrm{RMC}}(I)) = \sum_{j=1}^{n}\sum_{i=1}^{j-1} c_{ij} \cdot \Pr[\mathrm{sgn}(\widetilde{x^i} \cdot r) \neq \mathrm{sgn}(\widetilde{x^j} \cdot r)] \text{ , wobei}$$

$r$ zufällig und gleichverteilt gewählt ist. Nun genügt es zu zeigen, dass

$$\Pr[\mathrm{sgn}(\widetilde{x^i} \cdot r) \neq \mathrm{sgn}(\widetilde{x^j} \cdot r)] = \frac{\arccos(\widetilde{x^i} \cdot \widetilde{x^j})}{\pi} \text{ ist.}$$

Für die Funktion sgn gilt, dass $\mathrm{sgn}(\widetilde{x^i} \cdot r) \neq \mathrm{sgn}(\widetilde{x^j} \cdot r)$ ist, genau dann wenn die zugehörige Zufallslinie $\ell$, senkrecht zu $r$, gerade $\widetilde{x^i}$ und $\widetilde{x^j}$ trennt. Seien $s$ und $t$ die Schnittpunkte von $\ell$ mit dem Einheitskreis um den Ursprung. Die Punkte $\widetilde{x^i}$ und $\widetilde{x^j}$ werden genau dann von $\ell$ getrennt, wenn entweder $s$ oder $t$ auf dem kürzeren Kreisbogen zwischen $\widetilde{x^i}$ und $\widetilde{x^j}$ liegt, siehe Abbildung 8.2 zur Illustration. Die Wahrscheinlichkeit, dass $s$ oder $t$ auf diesem Kreisbogen der Länge $\arccos(\widetilde{x^i} \cdot \widetilde{x^j})$ liegen, ist

$$\frac{\arccos(\widetilde{x^i} \cdot \widetilde{x^j})}{2 \cdot \pi} + \frac{\arccos(\widetilde{x^i} \cdot \widetilde{x^j})}{2 \cdot \pi} = \frac{\arccos(\widetilde{x^i} \cdot \widetilde{x^j})}{\pi} \text{ .}$$

Beachte, dass nicht sowohl $s$ als auch $t$ auf diesem Kreisbogen liegen können. □

Daraus kann nun eine Gütegarantie gefolgert werden.

**Satz 8.19.** *Für eine Instanz $I$ von* MaxCut *berechnet* Random MaxCut *eine Lösung mit dem Wert $C_{\mathrm{RMC}}(I)$, für die gilt*

$$\frac{E(C_{\mathrm{RMC}}(I))}{OPT(I)} \geq 0,8785 \text{ .}$$

*Beweis.* Definiere

$$\beta := \min_{0<\alpha\leq\pi} \frac{2\alpha}{\pi(1-\cos\alpha)} \text{ .}$$

Sei $\tilde{x}^i, \ldots, \tilde{x}^n$ eine optimale Lösung von $\mathrm{QP}^2(I)$ mit dem Wert

$$C(\mathrm{QP}^2(I)) = \frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{j-1} c_{ij}(1 - \tilde{x}^i \cdot \tilde{x}^j) \text{ .}$$

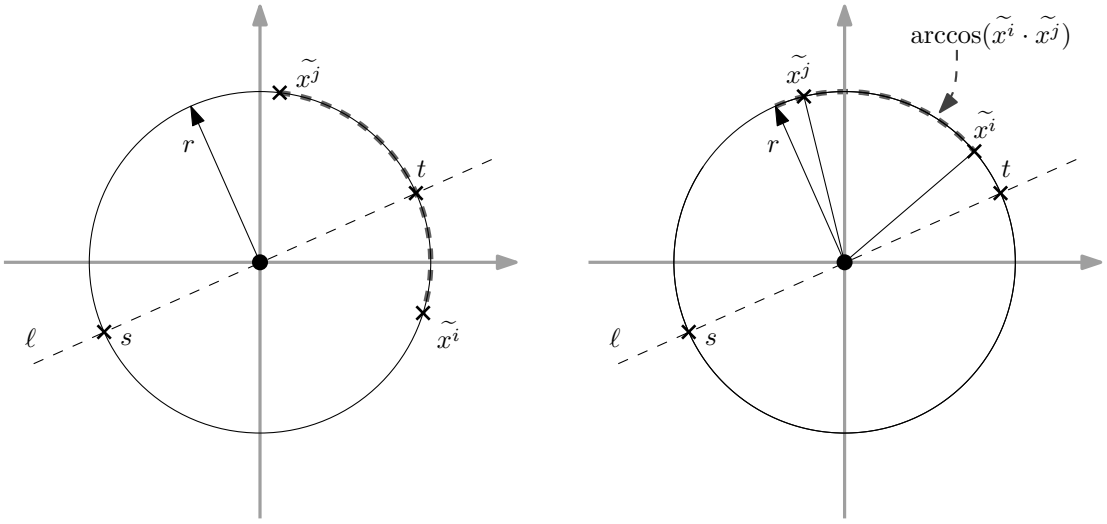



Abbildung 8.2: Runden der zweidimensionalen Lösung: Der kürzere Kreisbogen zwischen $\widetilde{x^i}$ und $\widetilde{x^j}$ ist grau gestrichelt.

Wenn $\alpha_{ij} := \arccos(\tilde{x}^i \cdot \tilde{x}^j)$, dann ist also $\cos \alpha_{ij} = \tilde{x}^i \cdot \tilde{x}^j$. Per Definition von $\beta$ ist dann

$$\beta \leq \frac{2 \cdot \alpha_{ij}}{\pi(1 - \cos \alpha_{ij})} = \frac{2}{\pi} \cdot \frac{\arccos(\tilde{x}^i \cdot \tilde{x}^j)}{1 - (\tilde{x}^i \cdot \tilde{x}^j)}, \text{ d.h.}$$

$$\frac{\arccos(\tilde{x}^i \cdot \tilde{x}^j)}{\pi} \geq \frac{\beta}{2} \cdot (1 - \tilde{x}^i \cdot \tilde{x}^j) \ .$$

Da das $\mathrm{QP}^2(I)$ eine Relaxierung des $\mathrm{IQP}(I)$ ist, gilt

$$\begin{aligned} E[C_{\mathrm{RMC}}(I)] &\geq \frac{1}{2} \cdot \beta \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij}(1 - \tilde{x}^i \cdot \tilde{x}^j) \\ &= \beta \cdot C(\mathrm{QP}^2(I)) \geq \mathrm{IQP}(I) \cdot \beta \\ &\geq \mathrm{OPT}(I) \cdot \beta \ . \end{aligned}$$

Man kann leicht zeigen, dass $\beta > 0.8785$ ist. □

**Effiziente Lösung von $\mathrm{QP}^2(I)$**

Random MaxCut ist polynomial, falls Schritt 2 polynomial ist. Es ist derzeit nicht bekannt, ob $\mathrm{QP}^2$ mit polynomialer Laufzeit gelöst werden kann. Man kann allerdings $\mathrm{QP}^2$ so modifizieren, dass Random MaxCut polynomial wird.

Ersetze $\mathrm{QP}^2$ durch folgendes $n$-dimensionales QP:

$$\max \frac{1}{2} \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij} \cdot (x^i \cdot x^j)$$

unter den Nebenbedingungen

$$x^i \text{ und } x^j \text{ sind } n\text{-dimensionale, normierte Vektoren über } \mathbb{R}.$$

Die Vektoren $x^i$ und $x^j$ erfüllen gewisse Bedingungen, die polynomiale Lösbarkeit garantieren. Doch zunächst benötigt man noch einige Begriffe.

**Definition 8.20.** *Eine $n \times n$-Matrix $M$ heißt* positiv semidefinit*, falls für jeden Vektor $x \in \mathbb{R}^n$ gilt:*

$$x^T \cdot M \cdot x \geq 0$$

Es ist bekannt, dass eine symmetrische Matrix $M$ genau dann positiv semidefinit ist, wenn es eine $m \times n$-Matrix $P$ ($m \leq n$) gibt, so dass

$$M = P^T \cdot P \tag{8.1}$$

ist. $P$ kann in polynomialer Laufzeit berechnet werden, falls $M$ positiv semidefinit ist.

Betrachte nun Vektoren $x^1, \ldots, x^n \in \mathbb{R}^n$ mit Norm 1 und definiere $M := (m_{ij})$ mit $m_{ij} := x^i \cdot x^j$. Dann ist $M$ wegen der Aussage zu Gleichung (8.1) positiv semidefinit. Andererseits gilt für jede positiv semidefinite $n \times n$-Matrix $M$ mit $m_{ii} = 1$ für $1 \leq i \leq n$, dass $n$ normierten Vektoren $x^1, \ldots, x^n \in \mathbb{R}^n$ mit $m_{ij} = x^i \cdot x^j$ in polynomialer Zeit berechnet werden können. Nun ist QP$^2$ äquivalent zu

$$\max \frac{1}{2} \cdot \sum_{j=1}^{n} \sum_{i=1}^{j-1} c_{ij} \cdot (1 - m_{ij}) \text{ , wobei}$$

$M = (m_{ij})$ eine positiv semidefinite Matrix ist und $m_{ii} = 1$ ist für $1 \leq i \leq n$. Dieses Problem heißt Semi-Definit-Cut($I$) oder SD-Cut($I$). Man kann nun beweisen, dass es für jedes $\epsilon > 0$ einen polynomialen Algorithmus $\mathcal{A}_\epsilon$ gibt mit

$$\mathcal{A}_\epsilon(I) \geq \mathrm{OPT}_{\text{SD-Cut}}(I) - \epsilon \ .$$

Dabei sei $\mathrm{OPT}_{\text{SD-Cut}}$ der optimale Lösungswert von SD-Cut. Der Algorithmus $\mathcal{A}_\epsilon$ ist polynomial in der Eingabegröße von $I$ und in $\log(\frac{1}{\epsilon})$. Daraus kann man folgern, dass $\mathcal{A}_\epsilon$ ein polynomialer, exakter Algorithmus für SD-Cut ist. Man kann nun zeigen, dass mit $\epsilon = 10^{-5}$ die Approximationsgarantie von $0,8785$ für Random MaxCut erreicht werden kann.

# Kapitel 9

# Parallele Algorithmen

**Der Literaturtip.** Zu parallelen Algorithmen vgl. [9].

Beim Entwurf paralleler Algorithmen ist in stärkerem Maß als bei sequentiellen Algorithmen das zugrundeliegende Berechnungsmodell von besonderer Bedeutung. Das Hauptmerkmal paralleler Algorithmen ist, dass anstatt eines Prozessors mehrere Prozessoren gleichzeitig, mehr oder weniger unabhängig voneinander, Operationen ausführen können. Mögliche Berechnungsmodelle, und auch wirkliche Parallelrechner, unterscheiden sich etwa darin, wie unabhängig die Prozessoren voneinander sind (gleichartige bzw. unterschiedliche Operationen in einem parallelen Schritt; Kommunikation kostet mehr oder weniger viel Berechnungszeit, etc.)

Ein sinnvolles Berechnungsmodell für den Entwurf von parallelen Algorithmen auf relativ hohem Abstraktionsniveau und für theoretische Betrachtungen wie etwa Komplexitätsbetrachtungen ist die PRAM (Parallel Random Access Machine), eine parallele Version der RAM (siehe Einführung). Das Konzept einer PRAM, das am meisten zitiert und für Algorithmen zugrundegelegt wird, stammt von Fortune & Wyllie (1978).

## 9.1 Das PRAM Modell

Das PRAM Modell ist festgelegt durch

- unbegrenzte Prozessorenzahl;
- unbegrenzten globalen Speicher, auf den alle Prozessoren Zugriff haben;
- Speicherzellen des globalen Speichers wie bei RAM;
- Operationen, die jeder einzelne Prozessor ausführen kann wie bei RAM.
- Jeder Prozessor hat einen eigenen lokalen Speicher, der ebenfalls unbegrenzt ist.

Jeder Prozessor darf nur auf seinen eigenen lokalen Speicher zugreifen, nicht aber auf den lokalen Speicher eines anderen Prozessors. Alle Prozessoren dürfen auf den globalen Speicher zugreifen.

**Problem.** *Was passiert, wenn mehrere Prozessoren gleichzeitig aus derselben Speicherstelle des globalen Speichers lesen wollen oder dieselbe Speicherzelle des globalen Speichers beschreiben wollen? Alle der folgenden Kombinationen sind denkbar und werden als Modelle untersucht bzw. beim Algorithmus zugrundegelegt:*

| *gleichzeitiges Lesen erlaubt* | *gleichzeitiges Lesen verboten* |
|---|---|
| *CR (*concurrent read*)* | *ER (*exclusive read*)* |
| *gleichzeitiges Schreiben erlaubt* | *gleichzeitiges Schreiben verboten* |
| *CW (*concurrent write*)* | *EW (*exclusive write*)* |

Wir wollen uns hier auf die CREW-PRAM beschränken.

## 9.2 Komplexität von parallelen Algorithmen

Wir betrachten wieder die Laufzeit im worst-case. Beim Ablauf eines parallelen Algorithmus werden $N$ Operationen, die gleichzeitig von $N$ Prozessoren ausgeführt werden, als ein (paralleler) Berechnungsschritt gezählt. Dann ist die *Laufzeit* $T_{\mathcal{A}}(n)$ eines parallelen Algorithmus $\mathcal{A}$ definiert als

**Definition 9.1.**

$$T_{\mathcal{A}}(n) := \max_{\substack{I \text{ Problembeispiel} \\ \text{der Größe } n}} \{\textit{Anzahl der Berechnungsschritte von } \mathcal{A} \textit{ bei Eingabe } I\}$$

**Bemerkung 9.2.** *Die Berechnung beginnt, wenn der erste Prozessor aktiv wird, also irgendeine Operation ausführt, und endet, nachdem der letzte Prozessor inaktiv geworden ist, also keine Operation mehr ausführt. Die Prozessoren arbeiten synchron.*

Für die Güte eines parallelen Algorithmus ist neben der Laufzeit (und dem Speicherplatzbedarf, den wir hier nicht betrachten) die Anzahl der benötigten Prozessoren relevant. Die *Prozessorenzahl* $P_{\mathcal{A}}(n)$ eines parallelen Algorithmus $\mathcal{A}$ ist definiert als

**Definition 9.3.**

$$P_{\mathcal{A}}(n) := \max_{\substack{I \text{ Problembeispiel} \\ \text{der Größe } n}} \left\{ \begin{array}{l} \textit{Anzahl an Prozessoren, die während des Ablaufs von } \mathcal{A} \\ \textit{bei Eingabe } I \textit{ gleichzeitig aktiv sind} \end{array} \right\}.$$

*(Dies entspricht also wieder einer worst-case-Abschätzung).*

Ein paralleler Algorithmus $\mathcal{A}$ steht natürlich in Konkurrenz zu sequentiellen Algorithmen. Dabei interessiert uns vor allem:

$$\text{speed-up}(\mathcal{A}) := \frac{\begin{array}{c}\text{worst-case Laufzeit des schnellsten}\\ \text{bekannten sequentiellen Algorithmus}\end{array}}{\begin{array}{c}\text{worst-case Laufzeit des}\\ \text{parallelen Algorithmus } \mathcal{A}\end{array}}$$

Je größer der speed-up($\mathcal{A}$) ist, umso besser ist natürlich der Algorithmus $\mathcal{A}$. Im Idealfall hofft man natürlich, einen speed-up von $N$ zu erreichen, wenn etwa $P_{\mathcal{A}}(n) =: N$ (für alle $n$). Bei PRAM-Algorithmen geht man allerdings meist davon aus, dass die Prozessorenzahl abhängig von der Problemgröße ist (wie aus der Definition 9.3 von $P_{\mathcal{A}}(n)$ ersichtlich). Man betrachtet also bei $T_{\mathcal{A}}$, $P_{\mathcal{A}}$, speed-up($\mathcal{A}$), usw. asymptotisches Verhalten.

Ein Qualitätsmaß für einen parallelen Algorithmus $\mathcal{A}$, in welches also sinnvoll Laufzeit und Prozessorenzahl eingehen, sind die *Kosten* $C_{\mathcal{A}}$ von $\mathcal{A}$:

$$C_{\mathcal{A}}(n) := T_{\mathcal{A}}(n) \cdot P_{\mathcal{A}}(n)$$

Sind asymptotisches Wachstum von $C_{\mathcal{A}}(n)$ und die schärfste asymptotische untere Schranke für die Laufzeit eines sequentiellen Algorithmus gleich, so nennt man $\mathcal{A}$ *kostenoptimal*. Kennt man keine gute untere Schranke, so betrachtet man die *Effizienz* $E_{\mathcal{A}}$ von $\mathcal{A}$:

$$E_{\mathcal{A}}(n) := \frac{\begin{array}{c}\text{worst-case Laufzeit des schnellsten}\\ \text{bekannten sequentiellen Algorithmus}\end{array}}{\text{Kosten des parallelen Algorithmus } \mathcal{A}}$$

Man kann davon ausgehen, dass $E_{\mathcal{A}}(n) \leq 1$, denn sonst könnte man aus $\mathcal{A}$ einen sequentiellen Algorithmus ableiten mit (asymptotisch) besserer Laufzeit, als die Laufzeit des schnellsten bekannten sequentiellen Algorithmus, indem man einen Prozessor alle Operationen eines parallelen Berechnungsschritts von $\mathcal{A}$ hintereinander ausführen lässt. Dann ist die Laufzeit $\approx C_{\mathcal{A}}(n)$.

## 9.3 Die Komplexitätsklassen

Die am meisten untersuchte Klasse in Zusammenhang mit parallelen Algorithmen ist die Klasse $\mathcal{NC}$:

**Definition 9.4 (Nick's Class nach Nicholas Pippinger).** *Die Klasse $\mathcal{NC}$ ist die Klasse der Probleme, die durch einen parallelen Algorithmus $\mathcal{A}$ mit polylogarithmischer Laufzeit und polynomialer Prozessorenzahl gelöst werden kann, d.h. $T_{\mathcal{A}}(n) \in ((\log_n)^{k_1})$ mit $k_1$ Konstante, und $P_{\mathcal{A}}(n) \in (n^{k_2})$ mit $k_2$ Konstante.*

Eine wichtige offene Frage ist: Gilt $\mathcal{P} = \mathcal{NC}$? Vermutung ist: „nein“. Wie bei der Frage, ob $\mathcal{P} = \mathcal{NP}$ ist, gibt es auch hier ein Vollständigkeitskonzept. Parallele Laufzeitkomplexität steht in engem Zusammenhang zu sequentieller Speicherplatzkomplexität.

**Definition 9.5 (Steve's Class nach Stephan Cook).** *Die Klasse $\mathcal{SC}$ ist die Klasse der Probleme, die durch einen sequentiellen Algorithmus mit polylogarithmischem Speicherplatzbedarf und polynomialer Laufzeit gelöst werden kann.*

Die offene Frage ist hier: $\mathcal{NC} = \mathcal{SC}$?

**Einschub.** Verschiedene PRAM-Modelle wie CREW-PRAM und EREW-PRAM, oder ein Modell, bei dem zu Beginn der Berechnung eine beliebige Anzahl von Prozessoren aktiv sind gegenüber einem Modell, bei dem zu Beginn nur ein Prozessor aktiv ist und alle anderen erst „aufgeweckt“

werden müssen, unterscheiden sich in der Laufzeit im wesentlichen nur um einen Faktor $\log N$ (bei $N$ Prozessoren, die gleichzeitig lesen wollen, bzw. aktiviert werden sollen). Dies lässt sich durch den folgenden Übertragungsalgorithmus, bei dem ein Wert an $N$ Prozessoren auf einer EREW-PRAM übertragen wird, veranschaulichen.

## 9.4 Parallele Basisalgorithmen

### 9.4.1 Broadcast

Das Ziel eines Broadcasts ist es, einen gegebenen Wert $m$ an alle $N$ Prozessoren zu übertragen. Eine formale Beschreibung ist in Algorithmus 53 gegeben. Dabei wird das EREW-PRAM Modell zugrundegelegt.

**Algorithmus 53** : BROADCAST($N$)

**Eingabe** : Wert $m$.
**Daten** : Array $A$ der Länge $N$ im globalen Speicher.

1 $P_1$ liest $m$, kopiert $m$ in den eigenen Speicher und schreibt $m$ in $A[1]$.
2 **Für** $i = 0$ **bis** $\lceil \log N \rceil - 1$
  `/* Mergewithhelp */`
3  **Für alle** $j : 2^i + 1 \leq j \leq 2^{i+1}$ **führe parallel aus**
4  $P_j$ liest $m$ aus $A[j - 2^i]$, kopiert $m$ in den eigenen Speicher und schreibt $m$ in $A[j]$

Die Laufzeit ist in $\mathcal{O}(\log N)$.

### 9.4.2 Berechnung von Summen

Ziel ist die Berechnung der Summe (bzw. Produkt, Minimum, Maximum, allgemein $n$-fache binäre Operation) aus $n$ Werten $a_1, \ldots a_n$.

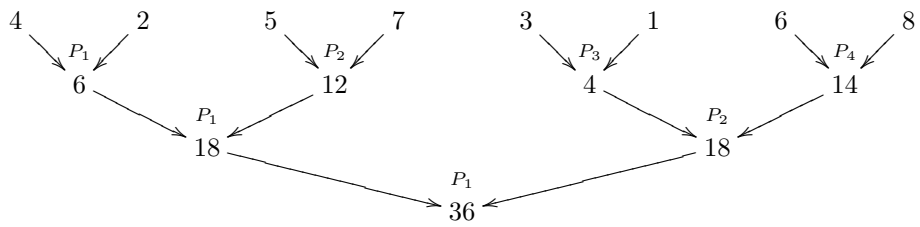
**Formale Beschreibung des Algorithmus** SUMME**.**

**Algorithmus 54** : SUMME($a_1, \ldots, a_n$)

**Eingabe** : $n$ Werte $a_1, \ldots, a_n$, o.B.d.A. sei $n = 2^m$.
**Ausgabe** : $\sum_{i=1}^{n} a_i$

1 **Für** $i = 1$ **bis** $m$
2  **Für alle** $j : 1 \leq j \leq \frac{n}{2^i}$ **führe parallel aus**
3  Prozessor $P_j$ berechnet $a_j := a_{2j-1} + a_{2j}$.
4 Gib $a_1$ aus.

**Beispiel: Die Berechnung einer Summe.**



Es ist $P_{\mathcal{A}}(n) = n/2$, $T_{\mathcal{A}}(n) \in \mathcal{O}(\log n)$, $C_{\mathcal{A}} \in \mathcal{O}(n \log n)$. Der Algorithmus $\mathcal{A}$ ist also nicht kostenoptimal.

**Bemerkung 9.6.** *Das logische ODER aus $n$ booleschen Variablen (0 und 1) kann ebenso auf einer CREW-PRAM in $\mathcal{O}(\log n)$ berechnet werden. Auf einer CRCW-PRAM könnte man mit $n$ Prozessoren das logische ODER aus $n$ Werten in $\mathcal{O}(1)$ bestimmen, wenn concurrent-write aufgelöst würde, indem mehrere Prozessoren an dieselbe Speicherzelle schreiben dürften g.d.w. sie denselben Wert schreiben wollen: Geht man davon aus, dass zu Beginn der Berechnung in allen Speicherzellen des globalen Speichers 0 steht, so hat das Ergebnis des logischen ODER den Wert 1 g.d.w. am Ende in Speicherzelle „1“ eine 1 steht.*

**Algorithmus 55** : ODER$(x_1, \ldots, x_n)$

```
1 Für alle i : 1 ≤ i ≤ n führe parallel aus
2     Prozessor P_i liest i-ten Eingabewert x_i
3     Wenn x_i = 1
4         P_i schreibt Wert 1 in die Speicherzelle „1“ des globalen Speichers.
```

Zurück zum Algorithmus $\mathcal{A}$ zur Berechnung von SUMME. Durch Rescheduling lässt sich aus $\mathcal{A}$ ein ein paralleler Algorithmus $\mathcal{A}'$ gewinnen, der Kosten $C_{\mathcal{A}'}(n) \in \mathcal{O}(n)$ hat. Dazu werden anstatt $n$ Prozessoren $\lceil \frac{n}{\log n} \rceil$ Prozessoren verwendet. Die $\frac{n}{2}$ Operationen, die im ersten Durchlauf von $\mathcal{A}$ von $\frac{n}{2}$ Prozessoren in einem parallelen Berechnungsschritt ausgeführt werden, werden stattdessen von $\lceil \frac{n}{\log n} \rceil$ in höchstens $\frac{\log n}{2}$ parallelen Berechnungsschritten ausgeführt, bzw. allgemein im $i$-ten Durchlauf $\frac{n}{2^i}$ Operationen in höchstens $\frac{\log n}{2^i}$ Berechnungsschritten. Damit ergibt sich insgesamt

$$T_{\mathcal{A}}(n) \leq c \cdot \sum_{i=1}^{\log n} \frac{\log n}{2^i} = c \cdot \log n \underbrace{\sum_{i=1}^{\log n} \frac{1}{2^i}}_{\leq 1} \in \mathcal{O}(\log n).$$

Es gilt $P_{\mathcal{A}'}(n) = \lceil n/\log n \rceil$, also $C_{\mathcal{A}'} \in \mathcal{O}(n)$.

### 9.4.3 Berechnung von Präfixsummen

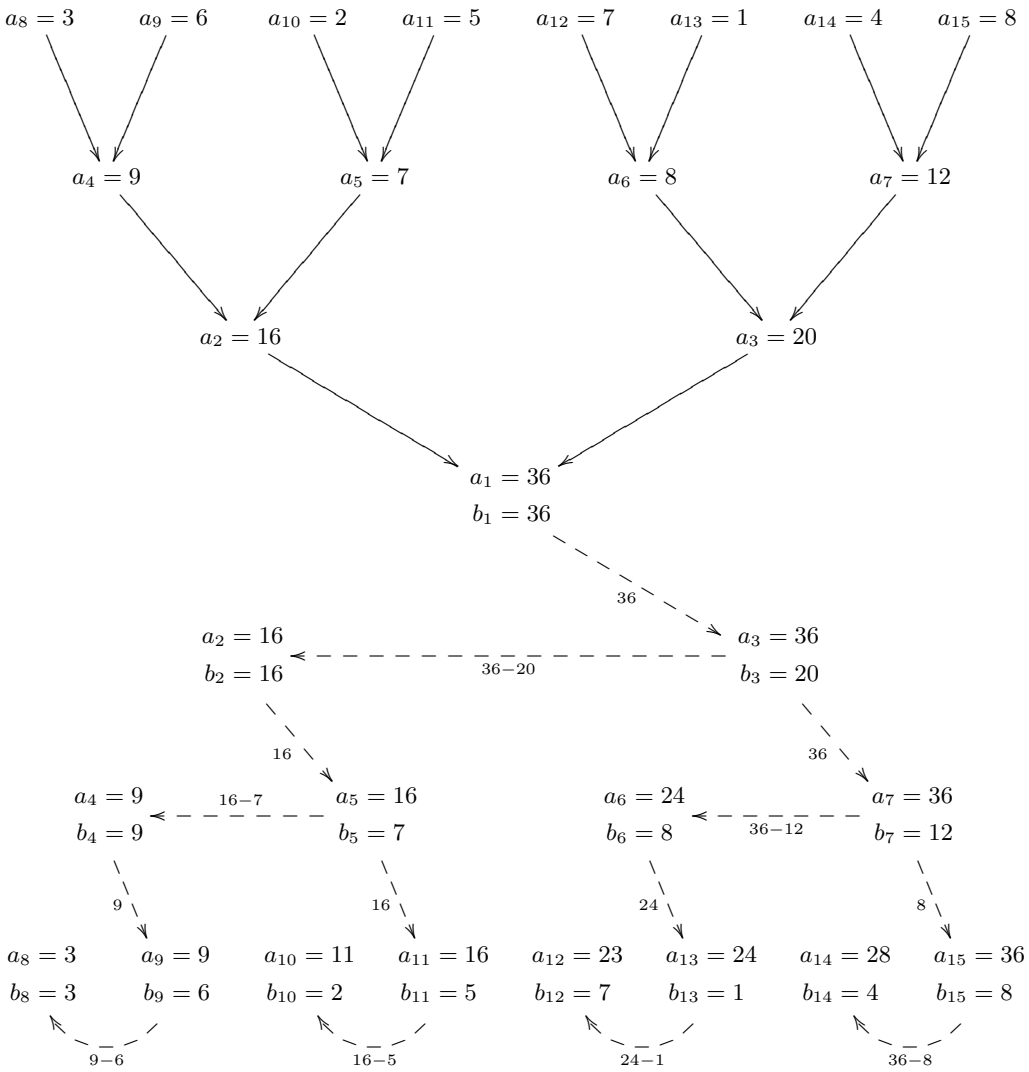
Für jedes $k$ mit $1 \leq k \leq n$ berechnen wir die Präfixsummen $A_k := \sum_{i=1}^{k} a_i$ aus den $n$ gegebenen Werten $a_1, \ldots, a_n$ (bzw. Produkt, Minimum, Maximum). Aus technischen Gründen nennen wir die Eingabewerte $a_n, a_{n+1}, \ldots, a_{2n-1}$; es sei wieder o.B.d.A. $n = 2^m$. Das Verfahren besteht aus zwei Phasen. In der ersten wird SUMME$(a_n, \ldots, a_{2n-1})$ ausgeführt. In der zweiten Phase werden aus dem Ergebnis von SUMME und aus Zwischenergebnissen des Verfahrens die $\sum_{i=n}^{k} a_i$ $(n \leq k \leq 2n-1)$ als Differenzen geeignet berechnet. Algorithmus 56 beschreibt das Vorgehen formal. Durch Induktion lässt sich leicht zeigen, dass $b_{2^j+\ell} = a_{2^j} + \ldots + a_{2^j+\ell}$ ist für $\ell + 0, \ldots, 2^j - 1$.

**Komplexität.** Offensichtlich ist die Laufzeit in $\mathcal{O}(\log n)$. Die Prozessorenzahl ist in $\mathcal{O}(n)$, es sind maximal $n/2$ Prozessoren gleichzeitig aktiv. Durch Rescheduling kann die Prozessorenzahl wieder auf $\mathcal{O}(n/\log n)$ bei Laufzeit $\mathcal{O}(\log n)$ reduziert werden, womit die Kosten wieder optimal sind.

**Algorithmus 56** : PRÄFIXSUMME($a_n, \ldots, a_{2n-1}$)

**Eingabe** : $n$ Werte $a_n, \ldots, a_{2n-1}$; $n = 2^m$.
**Ausgabe** : Die $n$ Präfixsummen $\sum_{i=1}^{k} a_i$ für $1 \leq k \leq n$.

```
1  Für j = m − 1 bis 0
2  |  Für alle i : 2^j ≤ i ≤ 2^{j+1} − 1 führe parallel aus
3  |  |  a_i := a_{2i} + a_{2i+1}
4  |  b_1 := a_1
5  Für j = 1 bis m
6  |  Für alle i : 2^j ≤ i ≤ 2^{j+1} − 1 führe parallel aus
7  |  |  Wenn i ungerade
8  |  |  |  b_i := b_{(i−1)/2}
9  |  |  sonst
10 |  |  |  b_i := b_{i/2} − a_{i+1}
```



### 9.4.4 Die Prozedur List Ranking oder Short Cutting

In einer linearen Liste (oder allgemeiner einem Wurzelbaum, in dem jedes Element $i$ nur seinen direkten Vorgänger kennt) sollen alle Elemente das erste Element in der Liste bzw. die Wurzel kennenlernen. Dies kann in $\mathcal{O}(\log n)$, wobei $n$ die Länge der Liste ist, bzw. in $\mathcal{O}(\log h)$, wobei $h$ die Höhe des Wurzelbaumes ist, mit $n$ Prozessen ($n$ Anzahl der Elemente im Baum) realisiert werden. Es sei Vor[root]=root.

**Formale Beschreibung des Algorithmus.**

**Algorithmus 57** : List Ranking($n$, $h$)

**Vorbedingung** : Array Vor[] enthält Vorgänger Vor[$i$] für jedes Element $1 \leq i \leq n$.
**Nachbedingung** : Array Vor[] enthält root für jedes Element $1 \leq i \leq n$.

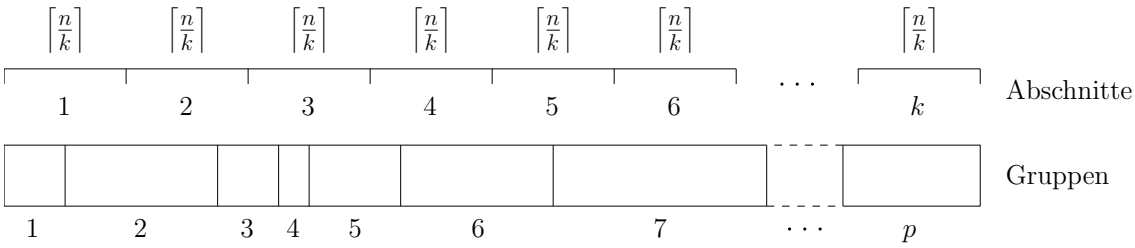
```
1 Für j = 1 bis ⌈log h⌉
2 |  Für alle i : 1 ≤ i ≤ n führe parallel aus
3 |  |  Vor[i] := Vor[Vor[i]]
```

### 9.4.5 Binäroperationen einer partitionierten Menge mit $K$ Prozessoren

**Problem.** *Gegeben sind $n$ Werte, die in $p$ Gruppen aufgeteilt sind. Mit $K$ Prozessoren sollen die $p$ Werte bestimmt werden, die sich als Binärverbindungen (also Summe, Minimum etc.) der Werte der $p$ Gruppen ergeben. Die Laufzeit soll $t(n)$ sein, mit*

$$t(n) \in \begin{cases} \lceil \frac{n}{K} \rceil - 1 + \log K, & \textit{falls } n > K \\ \log n, & \textit{sonst.} \end{cases}$$

Mit $K \geq n$ Prozessoren können die $p$ Binärverbindungen wie üblich in $(\log n)$ berechnet werden. Falls $K < n$ ist, teile die $n$ Werte in $K$ Abschnitte auf. Jeder Abschnitt erhält einen Prozessor.




Die Werte eines Abschnitts gehören entweder alle zu derselben Gruppe oder zu verschiedenen Gruppen. Gehören alle Werte zu derselben Gruppe, so wird die Binärverbindung aus diesen berechnet, ansonsten, falls sie zu $m$ Gruppen gehören, werden $m$ solche Binärverbindungen berechnet. Alle diese werden mit den $K$ Prozessoren jeweils sequentiell in höchstens $\lceil \frac{n}{K} \rceil - 1$ Schritten berechnet. Gehören alle Werte einer Gruppe zu demselben Abschnitt, so sind nach diesen $\lceil \frac{n}{K} \rceil - 1$ Schritten ihre Binärverbindungen endgültig berechnet. Für jeden Abschnitt sind jedoch höchstens 2 (bzw. 1) Ergebnisse noch nicht endgültig, sondern müssen mit anderen zusammengefasst werden. Falls also für jede Gruppe $G_i$ $(1 \leq i \leq p)$ die Anzahl der noch zusammenzufassenden Teilergebnisse $n_i$ ist, so gilt

$$\sum_{i=1}^{p} n_i \leq 2K - 2.$$

Ordne den Gruppen $G_i$ $(1 \leq i \leq p)$ jeweils $\lfloor \frac{n_i}{2} \rfloor$ Prozessoren zu, dann können diese in $\log n_i$ Zeit die endgültigen Ergebnisse bestimmen. Da $n_i \leq K$ gilt, ist dies in $\log K$. Die Anzahl an Prozessoren ist ausreichend, da

$$\sum_{i=1}^{p} \lfloor \tfrac{n_i}{2} \rfloor \leq \frac{1}{2} \cdot (2K - 2) < K.$$

## 9.5 Ein paralleler Algorithmus für die Berechnung der Zusammenhangskomponenten

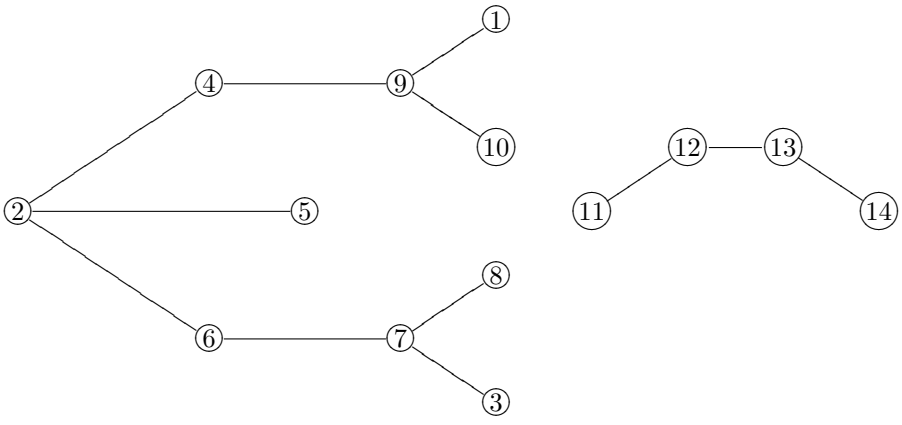
Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$. Bestimme durch einen CREW-PRAM-Algorithmus die Zusammenhangskomponenten von $G$.

*Idee.* Zunächst wird jeder Knoten als eine aktuelle Zusammenhangskomponente aufgefasst. Der Algorithmus besteht aus maximal $\lceil \log n \rceil$ Phasen, wobei in jeder Phase gewisse aktuelle Zusammenhangskomponenten zu neuen Zusammenhangskomponenten vereinigt werden. In einer Phase wird zunächst für alle Knoten parallel die aktuelle Zusammenhangskomponente (ungleich der eigenen) kleinster Nummer gewählt, die einen Nachbarknoten enthält. Dann wird für alle Knoten $i$ parallel die aktuelle Zusammenhangskomponente kleinster Nummer gewählt unter allen zuvor gewählten Zusammenhangskomponenten, die benachbart sind zu einem Knoten, der in derselben Komponente wie $i$ liegt. Jede Komponente kann nun mit der so bestimmten Zusammenhangskomponente kleinster Nummer zusammengefasst werden. Da bei diesem Schritt Ketten von Zusammenhangskomponenten entstehen können, muss in einer anschließenden Berechnung für alle Zusammenhangskomponenten, die zuvor zusammengefasst worden sind, eine gemeinsame neue Nummer bestimmt werden. Dies wird mit List Ranking realisiert. In jeder Phase wird die Zahl der aktuellen Zusammenhangskomponenten, die echte Subgraphen von Zusammenhangskomponenten von $G$ sind, halbiert. Daher endet der ALgorithmus tatsächlich nach maximal $\lceil \log n \rceil$ Phasen. □

**Der Algorithmus** Zusammenhang$(G)$ **(Chandra, Hirschberg & Sarwarte 1979)**

**Komplexität:** Die Laufzeit ist in $\mathcal{O}(\log^2 n)$: Schleife 3. hat Tiefe $\log n$ und innerhalb von 3. werden mehrere Minimum-Berechnungen vom Aufwand $\mathcal{O}(\log n)$ vorgenommen. 13. ist eine Ausführung von List Ranking und benötigt $\mathcal{O}(\log n)$. Die benötigte Prozessorenzahl ist zunächst $n^2$. Durch Rescheduling kann diese auf $\lceil \frac{n^2}{\log n} \rceil$ gesenkt werden, womit die Kosten in $\mathcal{O}(n^2 \log n)$ sind. Dies ist nicht kostenoptimal.

Wir betrachten folgenden Graphen:



**Algorithmus 58** : ZUSAMMENHANG($G$) [Chandra, Hirschberg & Sarwarte 1979]

**Eingabe** : $G = (V, E)$, $V = \{1, \ldots, n\}$
**Ausgabe** : Zu $i \in V$ steht in $K[i]$ kleinster Knoten, der in derselben Zusammenhangskomponente wie $i$ in $G$ liegt.
**Daten** :

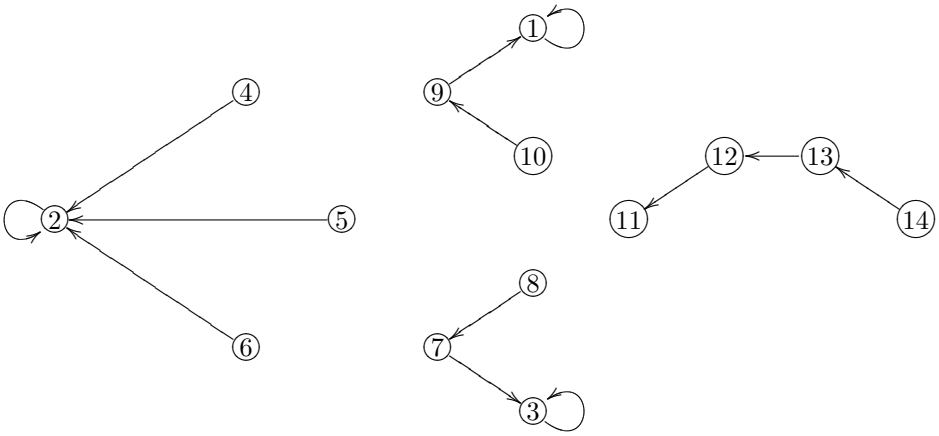
> Array $K$ der Länge $n$ ($K[i]$ enthält Nummer der aktuellen Zusammenhangskomponente, in der $i$ liegt.)

> Array $N$ der Länge $n$ ($N[i]$ enthält die Nummer der aktuellen Zusammenhangskomponente kleinster Nummer, in der ein Nachbar von $i$ bzw. eines Knoten aus $K[i]$ liegt).

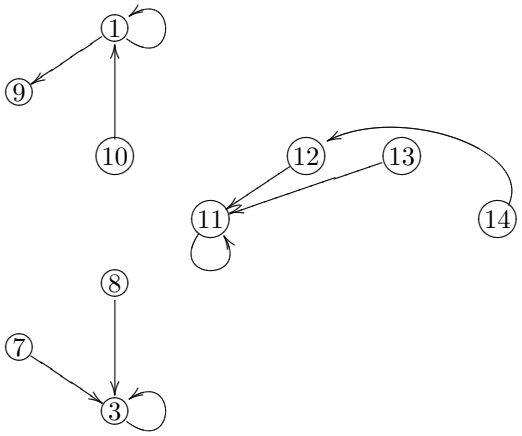
```
1  Für alle i : 1 ≤ i ≤ n führe parallel aus
2      K[i] := i
3  Für ℓ = 1 bis ⌈log n⌉
4      Für alle i : 1 ≤ i ≤ n führe parallel aus
5          Wenn ∃j : {i,j} ∈ E und K[i] ≠ K[j]
6              N[i] := min{K[j] : {i,j} ∈ E und K[i] ≠ K[j]}
7          sonst
8              N[i] := K[i]
9      Für alle i : 1 ≤ i ≤ n führe parallel aus
10         N[i] := min{N[j] : K[i] = K[j], N[j] ≠ K[j]}
11     Für alle i : 1 ≤ i ≤ n führe parallel aus
12         N[i] := min{N[i], K[i]}
13     Für alle i : 1 ≤ i ≤ n führe parallel aus
14         K[i] := N[i]
15     Für m = 1 bis ⌈log n⌉
16         Für alle i : 1 ≤ i ≤ n führe parallel aus
17             K[i] := K[K[i]]
```

Dann ist $n = 14$, $\lceil \log n \rceil = 4$. Der Algorithmus ZUSAMMENHANG geht wie folgt vor:

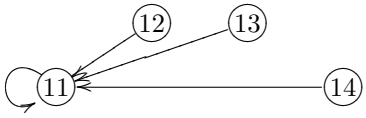
1. $K = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14]$
3. $\ell = 1$
4. $N = [9\ 4\ 7\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 12\ 11\ 12\ 13]$
9. $N = [9\ 4\ 7\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 12\ 11\ 12\ 13]$
11. $N = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 11\ 11\ 12\ 13]$
13. $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 11\ 11\ 12\ 13]$




15. $m = 1$
16. $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 3\ 1\ 1\ 11\ 11\ 11\ 12]$




15. $m = 2$
16. $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 3\ 1\ 1\ 11\ 11\ 11\ 11]$




15 Für $m = 3, 4$ ergibt sich keine Änderung mehr.
3. $\ell = 1$
4. $N = [1\ 2\ 3\ 1\ 2\ 3\ 2\ 3\ 2\ 1\ 11\ 11\ 11\ 11]$
9. $N = [2\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 11\ 11\ 11\ 11]$
11. $N = [1\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 1\ 1\ 11\ 11\ 11\ 11]$
13. $K = [1\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 1\ 1\ 11\ 11\ 11\ 11]$
15. $m = 1$
16. $K = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 11\ 11\ 11\ 11]$
15. Für $m = 2, 3, 4$ ergibt sich keine Änderung mehr.
3. Für $\ell = 3, 4$ ergibt sich keine Änderung mehr.

## 9.6 Modifikation zur Bestimmung eines MST

Der Algorithmus zur Bestimmung der Zusammenhangskomponenten kann geeignet modifiziert werden, so dass er in $(\log^2 n)$ Zeit mit $n^2$ bzw. $\lceil \frac{n^2}{\log n} \rceil$ Prozessoren einen MST in einem gewichteten

Graphen bestimmt.

### 9.6.1 Der Algorithmus

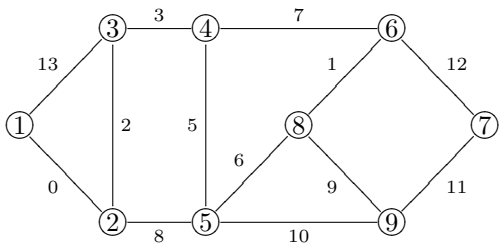
**Gegeben.** Graph $G = (V, E)$, $V = \{1, \ldots, n\}$ und Kantengewichte $c_{ij}$ für $(i, j) \in V \times V$, wobei $c_{ij} = \infty$, falls $\{i, j\} \notin E$. Zusätzlich werden im Algorithmus Datenstrukturen $T$ und $S$ benutzt. $T$ ist ein Array, das die zum MST gehörenden Kanten enthält. $S$ ist ein Array der Länge $n$, in dem an der Stelle $S[i]$ die Nummer desjenigen Knoten steht, der in derselben Komponente wie $i$ ist und eine Kante minimalen Gewichts zu einem Knoten einer anderen Komponente hat. In dem Array $N$ steht nun an der Stelle $N[i]$ die Nummer desjenigen Knotens, zu dem es von $i$ aus eine Kante minimalen Gewichts gibt und der in einer anderen Komponente als $i$ liegt.

Im Algorithmus ZUSAMMENHANG werden die Schritte 4. - 12. ersetzt. Der neue Algorithmus sieht damit aus wie folgt:

**Algorithmus 59** : MST($G$)

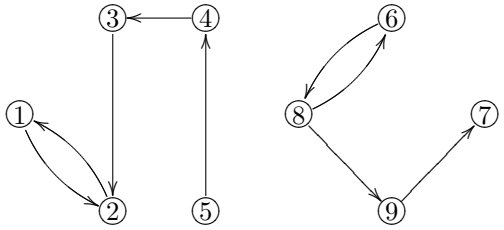
```
1  Für alle i : 1 ≤ i ≤ n führe parallel aus
2      K[i] := i
3  Für ℓ = 1 bis ⌈log n⌉
4      Für alle i : 1 ≤ i ≤ n führe parallel aus
5          Finde k ∈ V mit K[i] ≠ K[k] und c_ik = min{c_ij : 1 ≤ j ≤ n, K[i] ≠ K[j]}
6          N[i] := k
7      Für alle i : 1 ≤ i ≤ n führe parallel aus
8          Finde t ∈ V mit K[i] = K[t] und c_tN[t] = min{c_jN[j] : 1 ≤ j ≤ n, K[i] = K[j]}
9          setze N[i] := N[t] und S[i] := t
10     Für alle i : 1 ≤ i ≤ n führe parallel aus
11         Wenn N[N[i]] = S[i] und K[i] < K[N[i]]
12             setze S[i] := 0
13     Für alle i : 1 ≤ i ≤ n führe parallel aus
14         Wenn S[i] ≠ 0 und K[i] = i und c_N[i],S[i] ≠ ∞
15             setze T := T ∪ {N[i], S[i]}
16         Wenn S[i] ≠ 0
17             setze K[i] = K[N[i]]
18     Für m = 1 bis ⌈log n⌉
19         Für alle i : 1 ≤ i ≤ n führe parallel aus
20             K[i] := K[K[i]]
```

**Beispiel 9.7.** Wir betrachten folgenden Graphen:

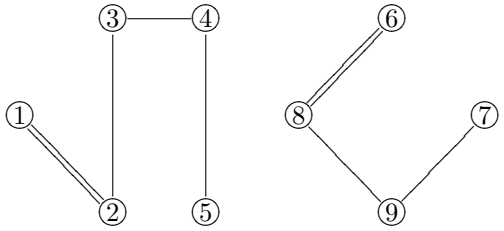



Der Algorithmus geht nun wie folgt vor:

1. $K := [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$
3. $\ell = 1$
4. $N := [2\ 1\ 2\ 3\ 4\ 8\ 9\ 6\ 8]$
7. $N := [2\ 1\ 2\ 3\ 4\ 8\ 9\ 6\ 8]$
   $S := [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$



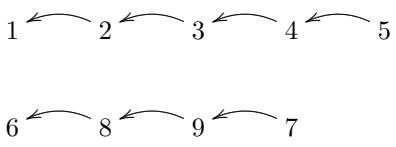

10. $S := [0\ 2\ 3\ 4\ 5\ 0\ 7\ 8\ 9]$
14. $T := \{\{1,2\},\{2,3\},\{3,4\},\{4,5\},\{6,8\},\{7,9\},\{8,9\}\}$
16. $K := [1\ 1\ 2\ 3\ 4\ 6\ 9\ 6\ 8]$



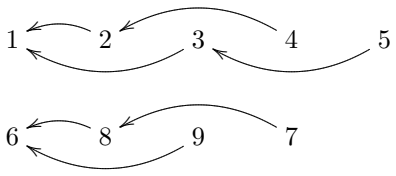

18. $m = 1$
19. $K := [1\ 1\ 1\ 2\ 3\ 6\ 8\ 6\ 6]$
18. $m = 2$ Die Schritte 18–20
19. $K := [1\ 1\ 1\ 1\ 1\ 6\ 6\ 6\ 6]$
18. Für $m = 3$ und $m = 4$ ergibt sich keine Änderung mehr.

lassen sich wie folgt darstellen:

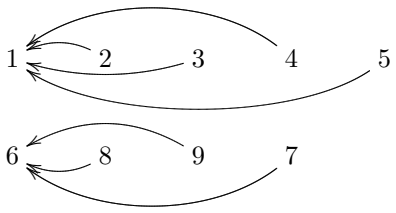
- $m = 0$.



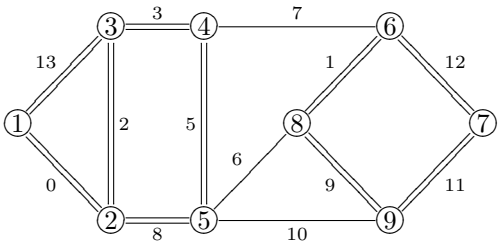

- $m = 1$.



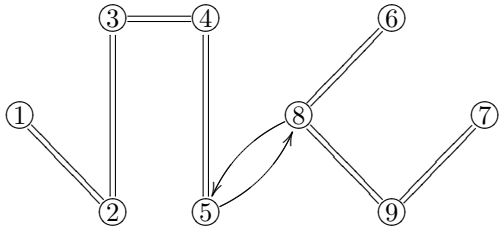

- $m = 2$.



- Ergebnis:

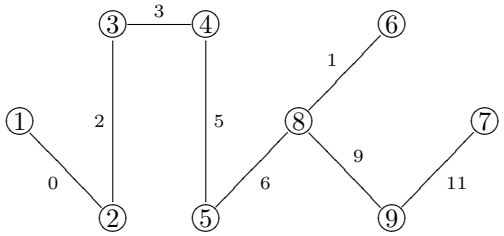



3. $\ell = 2$
4. $N := [6\ 6\ 6\ 6\ 8\ 4\ 1\ 5\ 5]$
7. $N := [8\ 8\ 8\ 8\ 8\ 5\ 5\ 5\ 5]$
   $S := [5\ 5\ 5\ 5\ 5\ 8\ 8\ 8\ 8]$



10. $S := [0\ 0\ 0\ 0\ 0\ 8\ 8\ 8\ 8]$
14. $T := [\{1,2\}\{2,3\}\{3,4\}\{4,5\}\{5,8\}\{6,8\}\{7,9\}\{8,9\}]$
16. $K := [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$
18. Für $m = 1$ ergibt sich keine Veränderung mehr.

**Endergebnis:**




### 9.6.2 Reduzierung der Prozessorenzahl auf $\frac{n^2}{\log^2 n}$

*Idee.* Schritt 4. bis 15. müssen jeweils nur für die Repräsentanten der Komponenten ausgeführt werden, nicht für alle Knoten. Die Anzahl der Komponenten halbiert sich jedoch mindestens in jeder Phase. Bei Vorgabe von $\lceil \frac{n^2}{\log^2 n} \rceil$ Prozessoren wird der Algorithmus an zwei Stellen modifiziert:

1. Minimum in Schritt 4. und 7. wird mit nur $\lceil \frac{n}{\log^2 n} \rceil$ Prozessoren ausgeführt;

2. nach jeder Phase wird die Adjazenzmatrix des Graphen neu aufgestellt, d.h. für alle Knoten $i$ und $j$ werden die Kanten $\{i, j\}$ auf die Repräsentanten der Komponenten von $i$ bzw. $j$ übertragen. □

**Realisierung von 1.** Binäroperation aus $n$ Werten mit $K$ Prozessoren.

**Problem.** *Gegeben seien $n$ Werte $a_1, \ldots, a_n$. Es soll die Summe (bzw. Produkt, Minimum, etc) aus $a_1, \ldots, a_n$ mit $K$ Prozessoren in Zeit $\mathcal{O}(t(n))$ bestimmt werden, wobei*

$$t(n) \leq \begin{cases} \lceil n/K \rceil - 1 + \log K, & \textit{falls } \lceil n/2 \rceil > K \\ \log n, & \textit{sonst.} \end{cases}$$

Für $K \geq \lceil n/2 \rceil$ ist das, wie bereits bewiesen, in $\mathcal{O}(\log n)$ Zeit möglich.

Sei $K < \lceil n/2 \rceil$: Teile $a_1, \ldots, a_n$ in $K$ Gruppen auf, wobei jede Gruppe $\lceil n/K \rceil$ bzw. die letzte Gruppe $n - (K-1) \cdot \lceil n/K \rceil$ Werte enthält. Jeder Gruppe wird ein Prozessor zugeteilt, der die Binärportion der Werte seiner Gruppe sequentiell in maximal $\lceil n/K \rceil - 1$ Schritten bestimmt. Die Binäroperation aus diesen $K$ Ergebnissen wird dann in $\log K$ Zeit mit $K$ Prozessoren parallel bestimmt.

Im Algorithmus ergibt sich dann eine Gesamtlaufzeit von

$$T(n) \in \mathcal{O}(()\frac{n}{k} + \log n \cdot \log k)$$

für die Schritte 4. und 7. mit $K \leq \lceil n/2 \rceil$ Prozessoren: Da sich die Anzahl der zu behandelnden Repräsentanten in jeder Phase halbiert und nach $\ell = \log n - \lceil \log K \rceil$ höchstens $2K$ Repräsentanten übrig sind, gilt:

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\ell-1} (\lceil \tfrac{n}{2^k \cdot K} \rceil - 1 + \log K) + \sum_{k=\ell}^{\log n - 1} (\log(\frac{n}{2^k}) \\ &\leq \frac{2n}{K} + \log n \cdot \log K - \log^2 K + \sum_{k=\ell}^{\log n - 1} \log n - k \\ &\in (\frac{n}{K} + \log n \cdot \log K). \end{aligned}$$

Die letzte Ungleichung gilt, da $\sum_{k=0}^{\infty} \frac{1}{2^k} \leq 2$.

**Realisierung von 2.** Um die Adjazenzmatrix des Graphen nach jeder Phase geeignet neu aufzustellen, werden die Knoten der Komponenten entsprechend der Nummer ihrer Repräsentanten angeordnet und jeweils an den Anfang einer solchen Knotengruppe ihr Repräsentant geschrieben. Zum Ermitteln der Einträge für alle Repräsentantenpaare muss jeweils für jede Knotengruppe die ODER-Verbindung der entsprechenden Einträge berechnet werden.

Dafür werden jeder Komponente $K = \lceil \frac{n}{\log^2 n} \rceil$ Prozessoren zugeordnet, um die ODER-Verbindungen für alle Adjazenzen von Knoten der Komponenten zu bilden. Dies ist eine Binärverbindung einer partitionierten Menge und kann mit $k$ Prozessoren in $\mathcal{O}(\lceil \frac{n}{k} \rceil - 1 + \log k)$ berechnet werden (siehe 9.4.5)

Analog wie bei 1. ist die Gesamtlaufzeit über alle Phasen für die Neuberechnungen der Adjazenzmatrix mit $K$ Prozessoren in $\mathcal{O}(()\frac{n}{K} + \log n \cdot \log K)$, also bei $K = \lceil \frac{n}{\log^2 n} \rceil$ in $\mathcal{O}(() \log^2 n)$.

## 9.7 ParallelSelect: Parallelisierung von Select zur Bestimmung des $k$-ten Elements aus $n$ Elementen

Das Verfahren ParallelSelect benutzt $p$ Prozessoren (wobei $p \leq n$) auf einer CREW-PRAM. Sei $S$ die Menge der gegebenen $n$ Elemente. Falls $|S| = n < 5$ ist, wird das $k$-te Element durch Sortieren bestimmt. Ansonsten wird $S$ in $p$ Teilfolgen der Länge höchstens $\lceil \frac{n}{p} \rceil$ aufgeteilt, und jeder

Prozessor bestimmt mittels Select in $\mathcal{O}(()\lceil\frac{n}{p}\rceil)$ das mittlere Element seiner Teilfolge. Beachte, dass Select wieder „ganz normal“ in Teilfolgen der Länge 5 unterteilt. Die mittleren Elemente bilden die Menge $M$, deren mittleres Element $m$ (d.h. das $\lceil\frac{|M|}{2}\rceil$-kleinste) durch einen rekursiven Aufruf von ParallelSelect bestimmt wird. $S$ wird dann wieder in Teilfolgen

$$S_< = \{x \in S : x \leq m\} \text{ und}$$
$$S_> = \{x \in S : x > m\}$$

geteilt. (Voraussetzung sei wieder, dass alle Elemente verschieden sind.) Falls $|S_<| \geq k$ ist, wird dann ParallelSelect rekursiv auf $S_<$ mit $k$ aufgerufen, ansonsten auf $S_>$ mit $k - |S_<|$. Da ParallelSelect kostenoptimal arbeiten soll, können $S_<$ und $S_>$ nicht durch sequentielles Durchlaufen der Menge $S$ und Vergleichen mit $m$ bestimmt werden. Stattdessen unterteilen die $p$ Prozessoren jeweils parallel ihre Teilfolgen $S^i$ $(1 \leq i \leq p)$ in

$$S^i_< = \{x \in S^i : x \leq m\} \text{ und}$$
$$S^i_> = \{x \in S^i : x > m\}.$$

$S_<$ und $S_>$ werden dann aus diesen $S^i_<$ und $S^i_>$ zusammengesetzt, indem die Prozessoren gleichzeitig in verschiedene Positionen eines Arrays $S_<$ im gleichen Speicher ihre $S^i_<$ schreiben. Die Positionen werden zuvor mittels Prefix-Summe berechnet.

**Formale Beschreibung von** ParallelSelect$(S, k; p)$

**Laufzeit.** Die Laufzeit von ParallelSelect ist in $(n^\ell)$, wobei $p = n^{1-\ell}$ die Anzahl der Prozessoren ist.

| | | |
|---|---|---|
| Schritt 1: | $\mathcal{O}(1)$ | |
| Schritt 4: | $\mathcal{O}(1)$ | |
| Schritt 6: | $\mathcal{O}(n^\ell)$ | |
| Schritt 7: | $\mathcal{O}(1)$ | |
| Schritt 8: | $\mathcal{O}(()n^\ell)$ | |
| Schritt 9: | $\mathcal{O}(1)$ | |
| Schritt 10: | $t(n^{1-\ell})$, | wobei $t(n)$ Laufzeit von ParallelSelect$(n)$ ist. |
| Schritt 12: | $\mathcal{O}(n^\ell)$ | |
| Schritt 13: | $\mathcal{O}(\log n^{1-\ell})$ | |
| Schritt 16: | $\mathcal{O}(n^\ell)$ | |
| Schritt 17: | $\mathcal{O}(1)$ | |
| Schritt 19: | $\leq t(\frac{3}{4}n + \frac{n^{1-\ell}}{4})$ | |
| Schritt 21–22: | $\leq t(\frac{3}{4}n + \frac{n^{1-\ell}}{4})$ | |

Die letzten beiden gelten wegen

$$S_< \leq n - \lceil\tfrac{n^{1-\ell}}{2}\rceil \cdot \lceil\tfrac{\lceil n^\ell\rceil+1}{2}\rceil \leq \frac{3}{4}n + \frac{n^{1-\ell}}{4}.$$

Denn wenn $m$ das $\lceil\frac{|M|}{2}\rceil$-kleinste Element von $M$ ist, so gibt es $\lceil\frac{|M|}{2}\rceil$ Elemente in $M$, die größer als $m$ sind, und für jedes dieser Elemente existieren mindestens $\lceil\frac{\lceil n^\ell\rceil-1}{2}\rceil$ Elemente in $S$, die größer sind. Damit ergibt sich für geeignete Konstanten $c_1$, $c_2$:

$$t(n) \leq c_1 \cdot n^\ell + t(n^{1-\ell}) + t(\frac{3}{4}n + \frac{n^{1-\ell}}{4}) + c_2 \cdot \log n^{1-\ell}.$$

Wähle Konstante $c_3$ so, dass $c_2 \cdot \log n^{1-l} \leq c_3 \cdot n^\ell$ ist, und setze $c_4 := c_3 + c_1$; dann ergibt sich per

**Algorithmus 60** : ParallelSelect$(S, k; p)$

**Eingabe** : $S$ mit $|S| = n$, $k$, $p$
**Daten** : Arrays $S_<$ und $S_>$ und $M$ im globalen Speicher.
Arrays $S^i$, $S^i_<$ und $S^i_>$ in den lokalen Speichern für $1 \le i \le p$.

**Wenn** $n < 5$
  Prozessor $p_1$ liest Folge $S$, sortiert $S$ und gibt das $k$-kleinste Element aus
**sonst**
  Prozessor $p_1$ berechnet aus $n$ und $p$ die Zahl $\ell$ mit $p = n^{1-\ell}$
  **Für alle** $i: \ 1 \le i \le p$ **führe parallel aus**
    Prozessor $p_i$ berechnet $(i-1) \cdot \lfloor n^\ell \rfloor$ und $i \cdot \lfloor n^\ell \rfloor - 1$ und kopiert die Elemente aus $S$, die zwischen Position $(i-1) \cdot \lfloor n^\ell \rfloor$ und Position $i \cdot \lfloor n^\ell \rfloor - 1$ stehen, nach $S^i$
    Prozessor $p_i$ berechnet $n' := n^{1-\ell} \cdot \lfloor n^\ell \rfloor$ und kopiert das Element aus $S$, das an Position $n' + i$ steht, nach $S^i$, falls vorhanden
    Prozessor $p_i$ führt Select$(S^i, \lceil \frac{1}{2} n^\ell \rceil)$ aus und schreibt das Ergebnis $m_i$ an die $i$-te Position von $M$
  Prozessor $p_1$ berechnet $p' := \lceil |M|^{1-\ell} \rceil$ (beachte $|M| = p = n^{1-\ell}$)
  ParallelSelect$(M, \lceil \frac{|M|}{2} \rceil; p')$ Ergebnis sei $m$
  **Für alle** $i: \ 1 \le i \le p$ **führe parallel aus**
    Prozessor $p_i$ berechnet $S^i_< = \{x \in S^i : x \le m\}$,$|S^i_<| =: a_i$
    $S^i_> = \{x \in S^i : x > m\}$,$|S^i_>| =: b_i$
    Präfix-Summe$(a_1, \ldots, a_p)$ Präfix-Summe$(b_1, \ldots, b_p)$. Die Ergebnisse seien $A_1, \ldots, A_p$ und $B_1, \ldots, B_p$
    Setze $A_0 = B_0 = 0$
  **Für alle** $i: \ 1 \le i \le p$ **führe parallel aus**
    Prozessor $p_i$ schreibt $S^i_<$ an die Positionen $(A_{i-1} + 1)$ bis $A_i$ von $S_<$ und $S^i_>$ an die Positionen $(B_{i-1} + 1)$ bis $B_i$ von $S_>$
  Prozessor $p_1$ berechnet aus $A_p = |S_<|$ die Zahl $s_1 := \lceil A_p^{1-\ell} \rceil$ und aus $B_p = |S_>|$ die Zahl $s_2 := \lceil B_p^{1-\ell} \rceil$
  **Wenn** $|S_<| \ge k$
    ParallelSelect$(S_<; k; s_1)$
  **sonst**
    Prozessor $p_1$ berechnet $k' := k - |S_<|$
    ParallelSelect$(S_>; k'; s_2)$

Induktion über $n$:

$$\begin{aligned} t(n) &\le c_4 \cdot n^\ell + c \cdot (n^{1-\ell})^\ell + c \cdot (\frac{3}{4} n + \frac{n^{1-\ell}}{4})^\ell \\ &\le c_4 \cdot n^\ell + c \cdot \frac{1}{n^{\ell^2}} \cdot n^\ell + c \cdot (\frac{3}{4} n)^\ell + c \cdot (\frac{n^{1-\ell}}{4})^\ell, \quad \text{da} \ell < 1 \\ &\le n^\ell (\frac{c_4}{c} + \frac{1}{n^{\ell^2}} + (\frac{3}{4})^\ell + \frac{1}{4^\ell \cdot n^{\ell^2}}) \\ &\le n^\ell (\frac{c_4}{c} + \frac{2}{n^{\ell^2}} + (\frac{3}{4})^\ell). \end{aligned}$$

Für festes $\ell$ $(0 \le \ell \le 1)$ ist $(\frac{3}{4})^\ell = 1 - \varepsilon$, wobei $\varepsilon > 0$ ist. Das bedeutet, für geeignetes $c$ und genügend großes $n$ ist

$$\frac{c_4}{c} + \frac{2}{n^{\ell^2}} + (\frac{3}{4})^\ell \le 1$$

und damit $t(n) \le c \cdot n^\ell$.

**Beispiel 9.8.** Sei $n = 17$, $p = 4$, $k = 7$. **Gesucht:** das 7.-kleinste Element.
**Vorgehen des Algorithmus.**

$S := [24\ \ 3\ \ 12\ \ 21\ \ 7\ \ 6\ \ 13\ \ 20\ \ 23\ \ 14\ \ 1\ \ 4\ \ 15\ \ 16\ \ 2\ \ 22\ \ 19]$

7.

$$\begin{aligned} S^1 &= [24\ \ 3\ \ 12\ \ 21\ \ 19] \\ S^2 &= [7\ \ 6\ \ 13\ \ 20] \\ S^3 &= [23\ \ 14\ \ 1\ \ 4] \\ S^4 &= [15\ \ 16\ \ 2\ \ 22] \end{aligned}$$

8. $M = [19\ \ 13\ \ 14\ \ 16]$

9. $p' = 2$

10. $m = 14$

11.

$$\begin{array}{ll} S^1_< = [3\ \ 12] & S^1_> = [24\ \ 21\ \ 19] \\ S^2_< = [7\ \ 6\ \ 13] & S^2_> = [20] \\ S^3_< = [14\ \ 1\ \ 4] & S^3_> = [23] \\ S^4_< = [2] & S^4_> = [15\ \ 16\ \ 22] \end{array}$$

16.

$$\begin{aligned} S_< &= [3\ \ 12\ \ 7\ \ 6\ \ 13\ \ 14\ \ 1\ \ 4\ \ 2] \\ S_> &= [24\ \ 21\ \ 19\ \ 20\ \ 23\ \ 15\ \ 16\ \ 22] \end{aligned}$$

17. $|S_<| = 9$, $|S_>| = 8$, $s_1 = 3$, $(s_2 = 3)$

Aufruf von ParallelSelect$(S_<; 7; 3)$

7.

$$\begin{aligned} S^{1,1} &= [3\ \ 12\ \ 7] \\ S^{1,2} &= [6\ \ 13\ \ 14] \\ S^{1,3} &= [1\ \ 4\ \ 2] \end{aligned}$$

8. $M^1$=[7 13 2]

10. $m = 7$

11.

$$\begin{array}{ll} S^{11}_< = [3\ \ 7] & S^{11}_> = [12] \\ S^{12}_< = [6] & S^{12}_> = [13\ \ 14] \\ S^{13}_< = [1\ \ 4\ \ 2] & \end{array}$$

16.

$$\begin{aligned} S^1_< &= [3\ \ 7\ \ 6\ \ 1\ \ 4\ \ 2] \\ S^1_> &= [12\ \ 13\ \ 14] \end{aligned}$$

20. $|S^1_<| = 6 < 7$, $k' = 7 - 6 = 1$

ParallelSelect mit $S^1_< = [12\ \ 13\ \ 14]$ und $k = 1$ liefert sofort das Ergebnis:

Das gesuchte 7.-kleinste Element ist die 12. ∎

## 9.8 Ein paralleler Algorithmus für das Scheduling-Problem für Jobs mit festen Start- und Endzeiten

**Problem.** *Gegeben seien $n$ Jobs $J_i$ mit Startzeiten $s_i$ und Endzeiten $t_i$ ($1 \leq i \leq n$), o.B.d.A. $s_i \neq s_j$, $t_i \neq t_j$ für $i \neq j$.*

*Gesucht ist ein optimaler Schedule, d.h. eine Zuordnung der $J_i$ auf einer minimalen Anzahl an Maschinen, so dass sich keine zwei Jobs auf derselben Maschine überlappen.*

Sequentiell kann das Problem in $\mathcal{O}(n \log n)$ Zeit wie folgt gelöst werden:

---

Sortiere die $s_1, t_1, s_2, t_2, \ldots, s_n, t_n$ in nichtabsteigender Reihenfolge, wobei $t_j$ vor $s_k$ kommt, falls $t_j = s_k$ Schreibe sortierte Folge in Array $U$ Setze $\ell := 1$ und $S[i] := i$ für $1 \leq i \leq n$ **Für** $k = 1$ **bis** $2n$

- **Wenn** $U[k] = s_j$
  - Setze $M[j] := S[\ell]$, $S[\ell] := 0$ und $\ell := \ell + 1$
- **sonst, wenn** $U[k] = t_j$
  - Setze $S[\ell - 1] := M[j]$ und $\ell := \ell - 1$

---

Dabei realisiert $S$ einen Stack, auf dem die freien Maschinen liegen; $M[j]$ gibt an, welcher Maschine Job $J_j$ zugeordnet wird.

Die parallele Version des Algorithmus besteht aus drei Phasen.

**1. Phase.** Die $s_i$, $t_i$ werden mittels eines parallelen Sortieralgorithmus sortiert und in ein Array $U$ der Länge $2n$ geschrieben. Für alle Jobs $J_i$ wird parallel die Anzahl $a_i$ der Maschinen berechnet, die unmittelbar nach der Startzeit von $J_i$ belegt sind sowie die Anzahl $b_i$ der Maschinen, die unmittelbar vor Beendigung von $J_i$ belegt sind. Dazu wird ein Array $L$ der Länge $2n$ angelegt mit

$$L[k] = \begin{cases} 1, & \text{falls } U[k] \text{ Startzeit} \\ -1, & \text{falls } U[k] \text{ Endzeit.} \end{cases}$$

Wenn nun $\sum_{j=1}^{k} L[j] = p_k$ für $1 \leq k \leq 2n$, dann ist

$$\begin{aligned} a_i &= p_k, && \text{falls } U[k] \text{ Startzeit } s_i \text{ und} \\ b_i &= p_k + 1, && \text{falls } U[k] \text{ Endzeit } t_i. \end{aligned}$$

**2. Phase.** Für alle parallelen Jobs $J_i$ wird der unmittelbare Vorgänger auf derselben Maschine berechnet (falls er existiert). Dies ist gerade der Job $J_\ell$, der als letzter Job vor bzw. zum Zeitpunkt $s_i$ geendet hat und für den $a_i = b_\ell$ ist.

**3. Phase.** Durch List Ranking wird der erste Vorgänger eines jeden Jobs $J_i$ auf derselben Maschine berechnet und den Jobs die entsprechende Maschinennummer zugeordnet.

Zum Schluss enthält $M[i]$ die Nummer der Maschine, der der Job $J_i$ zugeordnet wird. Die Schritte 3. bis 20. entsprechen der ersten Phase, die Schritte 21. bis 26. der zweiten Phase und 27. bis 31. der dritten Phase. Der Algorithmus benötigt $\mathcal{O}(\log n)$ Zeit bei $\frac{n^2}{\log n}$ Prozessoren, ist also nicht kostenoptimal.

**Algorithmus 61** : ParallelScheduling

**Eingabe** : $n$ Jobs $J_i$, Startzeiten $s_i$, Endzeiten $t_i$ $(1 \leq i \leq n)$
**Ausgabe** : $M[i]$ enthält Nummer der Maschine, der Job $J_i$ zugeordnet wird
**Daten** : Arrays $U$ und $L$ der Länge $2n$

**Für alle** $i: \ 1 \leq i \leq n$ **führe parallel aus**
  Setze $U[i] := s_i$ und $U[n+i] := t_i$
**Für alle** $i, j: \ 1 \leq i, j \leq 2n$ **führe parallel aus**
  **Wenn** $U[i] < U[j]$, *oder* $U[i] = U[j]$ *und* $U[i]$ *Endzeit,* $U[j]$ *Startzeit*
    Setze $r_{ij} := 1$
  **sonst**
    setze $r_{ij} := 0$
**Für alle** $j: \ 1 \leq j \leq 2n$ **führe parallel aus**
  Berechne $\Pi(j) := \sum_{i=1}^{2n} r_{ij}$
  setze $U[\Pi(j)+1] := U[j]$
**Für alle** $k: \ 1 \leq k \leq 2n$ **führe parallel aus**
  **Wenn** $U[k]$ *Startzeit*
    Setze $L[k] := 1$
  **sonst**
    setze $L[k] := -1$
  Berechne $p_k = \sum_{\ell=1}^{k} L[\ell]$
  **Wenn** $U[k] = s_j$
    Setze $a_j := p_k$
  **sonst, wenn** $U[k] = t_j$
    setze $b_j := p_k + 1$
**Für alle** $i: \ 1 \leq i \leq n$ **führe parallel aus**
  Finde $k$ mit $t_k = \max\{t_\ell : \ t_\ell \leq s_i, b_\ell = a_i\}$
  **Wenn** $k$ *existiert*
    Setze $\text{Vor}[i] := k$
  **sonst**
    setze $\text{Vor}[i] := i$
**Für** $\ell = 1$ **bis** $\lceil \log n \rceil$
  **Für alle** $i: \ 1 \leq i \leq n$ **führe parallel aus**
    Setze $\text{Vor}[i] := \text{Vor}[\text{Vor}[i]]$
**Für alle** $i: \ 1 \leq i \leq n$ **führe parallel aus**
  Setze $M[i] := a_{\text{Vor}[i]}$

**Formale Beschreibung des Algorithmus** Parallel Scheduling
**(Dekel & Sahni 1983)**

**Beispiel 9.9.**

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $s_i$ | 5 | 0 | 1 | 2 | 4 | 3 | 7 | 6 |
| $t_i$ | 8 | 4 | 5 | 7 | 9 | 6 | 10 | 12 |

**Vorgehen des Algorithmus.**

2. $U = [5\ 0\ 1\ 2\ 4\ 3\ 7\ 6\ 8\ 4\ 5\ 7\ 9\ 6\ 10\ 12]$

5.

$r_{ij}$ :

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 10 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 11 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

9. $\Pi = [7\ \ 0\ \ 1\ \ 2\ \ 5\ \ 3\ \ 11\ \ 9\ \ 12\ \ 4\ \ 6\ \ 10\ \ 13\ \ 8\ \ 14\ \ 15]$

7. $U = [0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 4\ \ 5\ \ 5\ \ 6\ \ 6\ \ 7\ \ 7\ \ 8\ \ 9\ \ 10\ \ 12]$

9. $L = [1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ -1]$

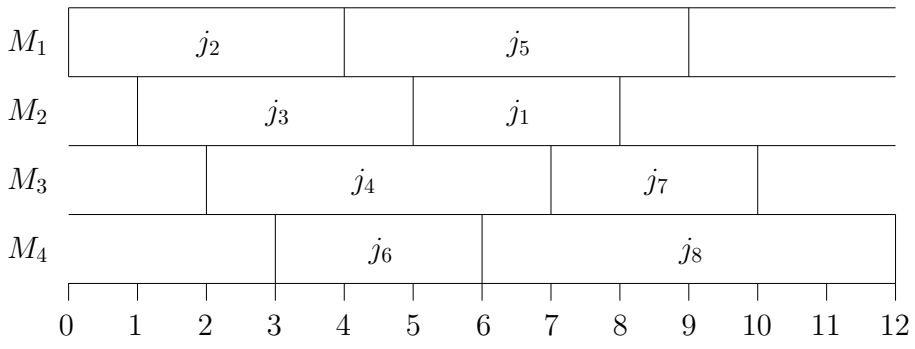
11.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $a_j$: | 4 | 1 | 2 | 3 | 4 | 4 | 4 | 4 |
| $b_j$: | 4 | 4 | 4 | 4 | 3 | 4 | 2 | 1 |

14. $\textsc{Vor} = [3\ \ 2\ \ 3\ \ 4\ \ 2\ \ 6\ \ 4\ \ 6]$

15. $\ell = 1$.


19.




■

# Kapitel 10

# Parametrisierte Algorithmen

> **Der Literaturtip.** Die algorithmischen Aspekte dieses Kapitels wie Kernbildung und beschränkte Suchbäume werden in einem einführenden Buch zu Parametrisierten Algorithmen von Niedermeier [16] beschrieben. Die Bücher von Downey und Fellows [6] sowie Flum und Grohe [7] behandeln hauptsächlich parametrisierte Komplexitätstheorie.

In diesem Kapitel betrachten wir die folgenden drei Probleme:

Independent Set

Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$

Gesucht: Unabhängige Menge $V' \subseteq V$ mit $|V'| \geq k$

Eine Menge $V' \subseteq V$ heißt *unabhängig* genau dann, wenn für alle $v, w \in V'$ gilt $\{v, w\} \notin E$ (siehe Abbildung 10.1).

Vertex Cover

Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$

Gesucht: Vertex Cover $V' \subseteq V$ mit $|V'| \leq k$

Eine Menge $V' \subseteq V$ heißt *Vertex Cover* genau dann, wenn für jede Kante $\{v, w\} \in E$ gilt $u \in V'$ oder $v \in V'$ (siehe Abbildung 10.1).

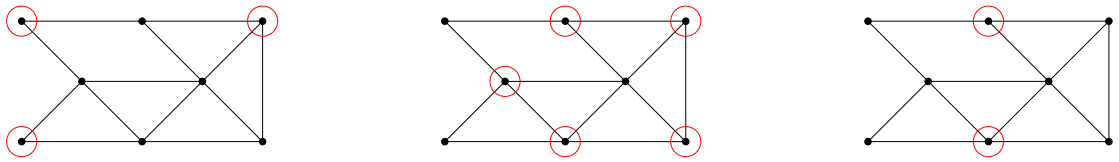
Dominating Set

Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$

Gesucht: Dominating Set $V' \subseteq V$ mit $|V'| \leq k$

Eine Menge $V' \subseteq V$ heißt *Dominating Set* genau dann, wenn für jeden Knoten $v \in V$ gilt, dass entweder $v$ selbst oder einer seiner Nachbarn in $V'$ enthalten ist (siehe Abbildung 10.1).

Diese Probleme sind $\mathcal{NP}$-schwer und lassen sich durch erschöpfendes Aufzählen (Brute-Force-Methode) aller $\binom{n}{k}$ Teilmengen $V' \subseteq V$ mit $|V'| = k$ mit einer Laufzeit in $\mathcal{O}(n^k \cdot (n + m))$ entscheiden.

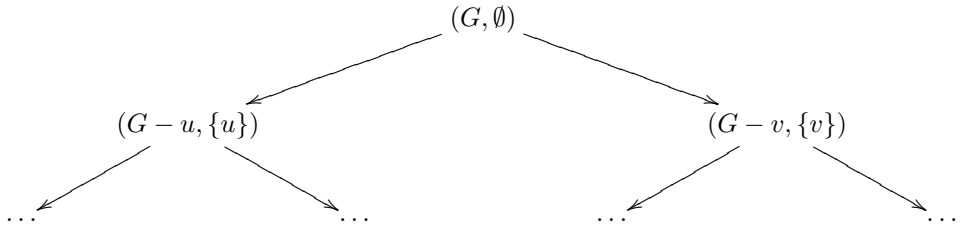


(a) Independent Set (b) Vertex Cover (c) Dominating Set

Abbildung 10.1: Independent Set, Vertex Cover und Domintating Set eines Beispiel-Graphen.

Im folgenden wird ein alternativer Algorithmus für Vertex-Cover vorgestellt. Es handelt sich dabei um einen parametrisierten Algorithmus, der auf einem tiefenbeschränkten Suchbaum basiert. Die Vorgehensweise des Algorithmus ist wie folgt:

1. Konstruiere vollständigen binären Baum der Tiefe $k$.
2. Markiere die Wurzel des Baumes mit $(G, \emptyset)$.
3. Jeder Knoten wird wie folgt rekursiv markiert: Sei $v$ ein mit $(H, S)$ markierter Knoten des Baumes, der zwei nicht markierte Kinder hat. Wähle eine beliebige Kante $\{u, v\}$ aus der Kantenmenge von $H$ und markiere das linke Kind mit $(H - u, S \cup \{u\})$ sowie das rechte Kind mit $(H - v, S \cup \{v\})$. Dabei bezeichne $H - v$ den Graphen, der durch Löschen des Knotens $v$ und aller inzidenten Kanten aus $H$ entsteht.
4. Falls es einen Baumknoten mit der Markierung $(H, S')$ existiert, wobei $H$ der leere Graph ist, dann ist $S'$ ein Vertex Cover der Größe $|V'| \leq k$. Andernfalls existiert kein Vertex Cover mit $k$ oder weniger Knoten für $G$.




Die Laufzeit bei diesem Ansatz ist in $\mathcal{O}(2^k \cdot (n + m))$ und damit von der Form $\mathcal{O}(\mathcal{C}(k) \cdot p(n))$, wobei $p$ ein Polynom und $\mathcal{C} : \mathbb{N} \to \mathbb{N}$ eine berechenbare Funktion ist, die nur von $k$ abhängt.

Ein Optimierungsproblem $\Pi$ kann in kanonischer Weise als *parametrisiertes Problem* aufgefasst werden: Zu einer Instanz $\mathcal{I}$ von $\Pi$ und einem Parameter $k \in \mathbb{N}$ wird eine zuklässige Lösung gesucht, deren Wert mindestens (Maximierungsproblem) bzw. höchstens (Minimierungsproblem) $k$ ist.

**Definition 10.1 (fixed parameter tractable).** *Ein parametrisiertes Problem $\Pi$ heißt* fixed parameter tractable*, wenn es einen Lösungsalgorithmus zu $\Pi$ mit Laufzeit $\mathcal{O}(\mathcal{C}(k) \cdot p(n))$ gibt. Dabei ist $n$ die Eingabegröße, $p$ ein Polynom, $k$ der Parameter und $\mathcal{C}$ eine berechenbare Funktion, die nur von $k$ abhängt.*

**Definition 10.2 (FPT).** *Die Klasse FPT ist die Klasse aller Probleme, die fixed parameter tractable sind.*

**Bemerkung 10.3.** Vertex Cover *ist in FPT. Für* Independent Set *und* Dominating Set *ist nicht bekannt, ob sie in FPT sind. Es wird jedoch vermutet, dass sie nicht in FPT sind.*

## 10.1 Parametrisierte Komplexitätstheorie (Exkurs)

**Definition 10.4 (Parametrisierte Reduktion).** *Eine* parametrisierte Reduktion *von einem parametrisierten Problem* $\Pi$ *auf ein zweites parametrisiertes Problem* $\Pi'$ *besteht aus einer Funktion* $f$*, die einer Instanz* $\mathcal{I}$ *und einem Parameter* $k$ *von* $\Pi$ *eine Instanz* $\mathcal{I}'$ *von* $\Pi'$ *zuordnet, sowie Funktionen* $\mathcal{C}', \mathcal{C}'' : \mathbb{N} \to \mathbb{N}$ *so, dass folgende Eigenschaften erfüllt sind:*

*(i)* $f((\mathcal{I}, k))$ *kann in* $\mathcal{O}(\mathcal{C}''(k) \cdot p(n))$ *berechenet werden, wobei* $n$ *die Eingabegröße von* $\mathcal{I}$ *und* $p$ *ein Polynom ist.*

*(ii)* $(\mathcal{I}, k)$ *ist eine Ja-Instanz von* $\Pi$ *ist genau dann, wenn* $(\mathcal{I}', \mathcal{C}'(k))$ *eine Ja-Instanz zu* $\Pi'$ *ist.*

**Bemerkung 10.5.** *(i) Es gibt eine Hierarchie von Kompexitätsklassen* $W[t]$*, die vermutlich echt ist, und FPT enthält*

$$FPT \subseteq W[1] \subseteq W[2] \cdots$$

*(ii) Mithilfe der parametrisierten Reduktion lässt sich ein Vollständigkeitsbegriff definieren, um die schweren Probleme in* $W[t]$ *zu identifizieren.*

*(iii)* Independent Set *ist* $W[1]$*-vollständig und* Dominating Set *ist* $W[2]$*-vollständig.*

## 10.2 Grundtechniken zur Entwicklung parametrisierter Algorithmen

In diesem Kapitel sollen zwei Grundtechniken zur Entwicklung parametrisierter Algorithmen vorgestellt werden:

1. *Kernbildung:* Dabei wird die Instanz durch Anwendung verschiedener Regeln auf einen *(schweren) Problemkern* reduziert.

2. *Tiefenbeschränkte Suchbäume:* Dabei wird eine erschöpfende Suche in einem geeigneten Suchbaum mit beschränkter Tiefe durchgeführt.

*Idee (zu 1.).* Reduziere Instanz $(\mathcal{I}, k)$ in Laufzeit $\mathcal{O}(p(\langle \mathcal{I} \rangle))$ auf eine äquivalente Instanz $\mathcal{I}'$, so dass die Größe $\langle \mathcal{I}' \rangle$ von $\mathcal{I}'$ nur von $k$ abhängt. Löse $\mathcal{I}'$ anschließend etwa durch erschöpfende Suche. □

**Beispiel 10.6 (Vertex Cover).** Um ein Vertex Cover $V'$ für $G = (V, E)$ mit höchstens $k$ Knoten zu bestimmen, kann man sich die folgenden Beobachtungen zunutze machen:

(i) Für alle $v \in V$ ist entweder $v$ selbst oder alle Nachbarn $N(v)$ in $V'$.

(ii) Für ein Vertex Cover $V'$ der Größe $k$ gilt

$$\{v \in V | N(v) > k\} \subseteq V',$$

d.h. ein solches Vertex Cover enthält alle Knoten mit Grad größer als $k$.

(iii) Falls $\Delta(G) \leq k$ und $|E| > k^2$ gelten, so hat $G$ kein Vertex Cover mit $k$ oder weniger Knoten. Dabei bezeichne $\Delta(G)$ den Maximalgrad von $G$. ■

Der folgende Algorithmus 62 von Buss nutzt diese Eigenschaften aus:

**Algorithmus 62** : VERTEX-COVER$(G, k)$

```
   Eingabe : Graph G = (V, E), Parameter k
1  H ← {v ∈ V | |N(v)| > k}
2  Wenn |H| > k
3  |  gib aus: G hat kein Vertex Cover der Größe k
4  sonst
5  |  k' ← k − |H|
6  |  G' ← G − H
7  |  Wenn |E(G')| > k · k'
8  |  |  gib aus: G hat kein Vertex Cover der Größe k
9  |  sonst
10 |  |  entferne alle isolierten Knoten aus G'
11 |  |  berechne ein Vertex Cover der Größe k' im verbleibenden Graphen
```

*Bemerkung zur Korrektheit:* Wenn der Graph $G'$ in Schritt 7/8 des Algorithmus kein Vertex Cover der Größe $k'$ hat, so hat $G$ kein Vertex Cover der Größe $k$. Falls $G'$ nun ein Vertex Cover der Größe $k'$ hat und der Maximalgrad von $G'$ höchstens $k$ ist, so kann jeder Knoten eines Vertex Covers in $G'$ höchstens $k$ Kanten überdecken. Es gilt dann also $|E(G')| \leq k \cdot k'$. Demnach hat $G'$ kein Vertex Cover der Größe $k'$, falls $|E(G')| > k \cdot k'$, und $G$ somit kein Vertex Cover der Größe $k$.

*Laufzeitanalyse:* Wenn der Algorithmus die Zeilen 10/11 erreicht, hat der Graph $G'$ maximal $k \cdot k'$ Kanten. Falls die Berechnung eines Vertex Covers der Größe $k'$ in Zeile 11 mit einer Laufzeit in $\mathcal{O}(f(k'))$ ausgeführt werden kann, so ist die Gesamtlaufzeit in $\mathcal{O}(nk + f(k'))$.

Nach Entfernen aller isolierten Knoten aus $G'$ hat der resultierende Graph höchstens $2k^2$ Knoten, da jede der höchstens $k^2$ Kanten höchstens zwei Knoten zur Knotenmenge beitragen kann. Es gibt somit höchstens $\binom{2k^2}{k'}$ Teilmengen mit $k'$ Elementen, die als Vertex Cover für $G'$ infrage kommen. Diese Anzahl läßt sich durch $(2 \cdot k^2)^k = \mathcal{O}(2^k \cdot k^{2k})$ abschätzen. Damit ergibt sich ein Gesamtaufwand in $\mathcal{O}(nk + 2^k \cdot k^{2k})$, wenn in Schritt 11 erschöpfende Suche verwendet wird.

Der Aufwand in Schritt 11 läßt sich jedoch durch Verwendung eines tiefenbeschränkten Suchbaums auch auf $\mathcal{O}(2^k \cdot k^2)$ reduzieren. Wenn sich die Größe des Suchbaums verringern lässt, kann man den Faktor $2^k$ eventuell noch weiter verbessern.

*Idee (zu 2.).* Der Aufwand, der durch Verwendung eines tiefenbeschränkten Suchbaumes entsteht, hängt wesentlich von der Struktur des Baumes ab. Im vorangegangenen Beispiel haben wir einen binären Suchbaum verwendet. Damit ergibt sich die Laufzeit durch die Abschätzung

$$T(k) = T(k-1) + T(k-1) + c\,. \tag{10.1}$$

Allgemein ergibt sich eine Rekursionsgleichung der Form

$$T(k) \leq T(k-t_1) + \cdots + T(k-t_s) + c\,. \tag{10.2}$$

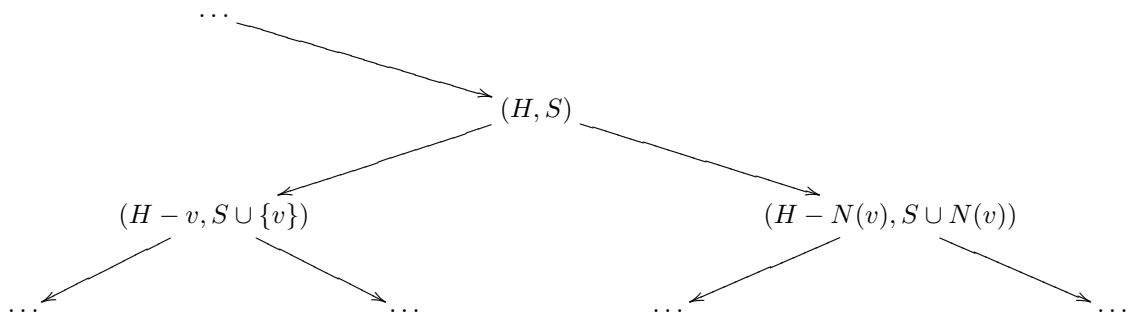
Der Vektor $(t_1, \ldots, t_s)$ heißt Verzweigungsvektor. Im obigen Beispiel ist der Verzweigungsvektor somit der Vektor $(1, 1)$. □

Mithilfe kombinatorischer Mittel (erzeugende Polynome) kann man das asymptotische Wachstum von $T(k)$ aus dem Verzweigungsvektor bestimmen. Für einige Verzweigungsvektoren ist die sich ergebende Basis in Tabelle 10.1 dargestellt.

**Beispiel 10.7.** Zur Illustration sei hier ein Beispiel mit Verzweigungsvektor $(1, 4)$ kurz skizziert: Für das Beispiel nehmen wir an, dass es immer einen Knoten $v$ mit $|N(v)| \geq 4$ gibt. In diesem Fall kann man verzweigen, indem man entweder $v$ oder alle Nachbarn von $v$ zum Vertex Cover hinzunimmt:

Tabelle 10.1: Basen ausgewählter Verzweigungsvektoren

| **Verzweigungsvektor** | **Basis** |
|---|---|
| $(1,1)$ | 2 |
| $(1,2)$ | 1.618 |
| $(1,3)$ | 1.4656 |
| $(1,4)$ | 1.3803 |
| $(1,5)$ | 1.325 |
| $(2,3)$ | 1.325 |
| $(3,3)$ | 1.26 |
| $(3,3,6)$ | 1.342 |
| $(3,4,6)$ | 1.305 |




Damit ergibt sich eine Rekursionsgleichung der Form

$$T(k) \leq T(k-1) + T(k-4) + c \tag{10.3}$$

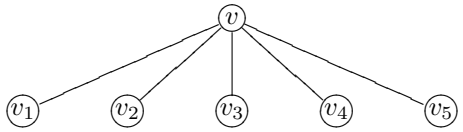
und daher eine Laufzeit von $\mathcal{O}(nk + 1.3803^k \cdot k^2)$ (siehe Tabelle 10.1). ∎

Man kann einen tiefenbeschränkten Suchbaum konstruieren, der zu einer Laufzeit von $\mathcal{O}(kn + 1.342^k \cdot k^2)$ führt, indem man an jedem Knoten mit Verzweigungsvektor $(1,5)$ oder $(2,3)$ oder $(3,2)$ oder $(3,3)$ oder $(3,4,6)$ oder $(3,3,6)$ (oder höchstens einmal $(1,4)$) verzweigt. Da auf jedem Pfad des Suchbaumes höchstens einmal mit Verzweigungsvektor $(1,4)$ verzweigt wird, wird die Laufzeit dominiert durch $(3,3,6)$ und es ergibt sich somit eine Laufzeit von $\mathcal{O}(kn + 1.342^k \cdot k^2)$.

*Idee.* Führe immer die Regel kleinster Nummer durch. Dadurch eliminiert man Knoten mit Grad $1, 5, 2, 3, (4)$. Sei $C$ die Menge der Knoten, die schon zu einem früheren Zeitpunkt in das Vertex Cover übernommen wurden. Jede der folgenden Regeln berechnet ein neues Vertex Cover $C'$, das sich aus $C$ ergibt.

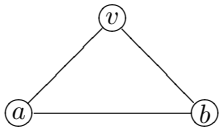
*Regel* 1: Es existiert ein Knoten $v$ mit Grad 1. In diesem Fall wird keine Verzweigung durchgeführt. Stattdessen wird der Nachbarknoten $w$ von $v$ in das Vertex Cover übernommen, d.h. $C' = C \cup \{w\}$.


*Regel* 2: Es existiert ein Knoten $v$ mit Grad $\geq 5$. In diesem Fall wird mit $(1,5)$ verzweigt, d.h. es wird entweder der Knoten $v$ oder seine mindestens fünf Nachbarn in das neue Vertex Cover $C'$ übernommen, d.h. $C' = C \cup \{v\}$ oder $C' = C \cup N(v)$.

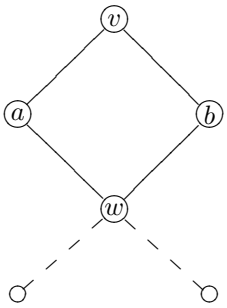



*Regel* 3: Es existiert ein Knoten $v$ mit Grad 2. Wir unterscheiden 3 Fälle:

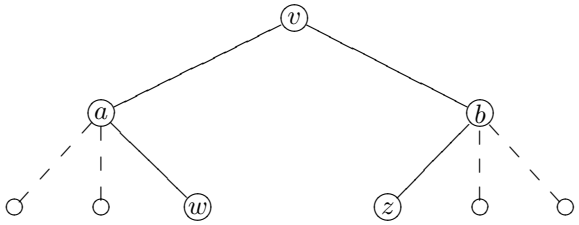
*Fall* 3.1: Die beiden Nachbarn $a$ und $b$ von $v$ sind adjazent. In diesem Fall werden $a$ und $b$ in das Vertex Cover übernomen, d.h. $C' = C \cup \{a, b\}$. Es muss nicht verzweigt werden, da $v, a, b$ ein Dreieck bilden und daher mindestens zwei Knoten im Vertex Cover enthalten sein müssen.




*Fall* 3.2: Die beiden Nachbarn $a$ und $b$ von $v$ sind nicht adjazent und sowohl $a$ als auch $b$ haben Grad zwei und sind mit einem gemeinsamen Nachbarn $w$ verbunden. In diesem Fall werden $v$ und $w$ in das Vertex Cover übernommen, d.h. $C' = C \cup \{v, w\}$.

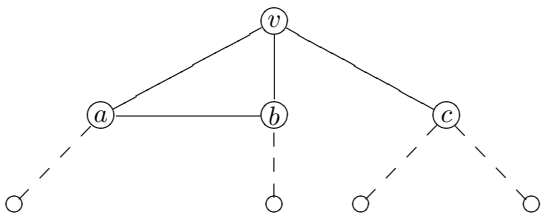



*Fall* 3.3: $|N(a) \cup N(b)| \geq 3$. In diesem Fall werden entweder $N(v)$ oder $N(a) \cup N(b)$ in das Vertex Cover übernommen, d.h. $C' = C \cup N(v)$ oder $C' = C \cup N(a) \cup N(b)$. Es wird also mit Verzweigungsvektor $(2, 3)$ verzweigt.

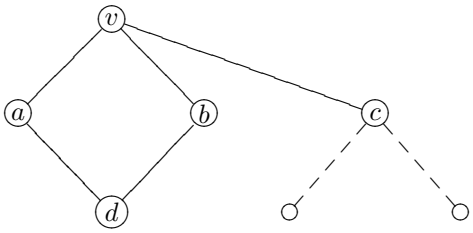



*Regel* 4: Es existiert ein Knoten $v$ mit Grad 3. Wir unterscheiden 4 Fälle.

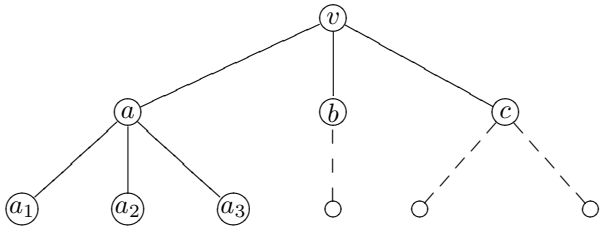
*Fall* 4.1: Der Knoten $v$ ist Teil eines Dreiecks $v, a, b$. Es werden entweder alle Nachbarn von $v$ oder alle Nachbarn von $c$ ins Vertex Cover übernommen, d.h. $C' = C \cup N(v)$ oder $C' = C \cup N(c)$. Man erhält einen Verzweigungsvektor von $(3, 3)$.



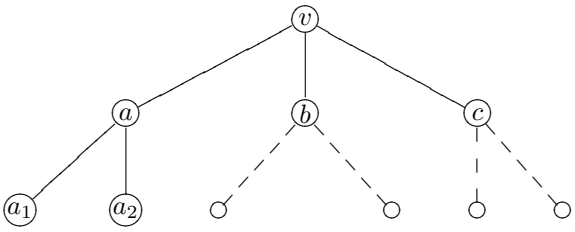
*Fall* 4.2: Der Knoten $v$ ist Teil eines Kreises der Länge 4 mit Knoten $v, a, b, d$. Es werden entweder alle Nachbarn von $v$ oder $c$ und $d$ in das Vertex Cover übernommen, d.h. $C' = C \cup N(v)$ oder $C' = C \cup \{v, d\}$. Man erhält einen Verzweigungsvektor von $(3, 2)$.




*Fall* 4.3: Es gibt keine Kante zwischen den Knoten $a, b, c$ und ohne Beschränkung der Allgemeinheit sei $|N(a)| \geq 4$. Es werden entweder alle Nachbarn von $v$ oder alle Nachbarn von $a$ oder $a$ sowie alle Nachbarn von $b$ und $c$ in das Vertex Cover übernommen, d.h. $C' = C \cup N(v)$ oder $C' = C \cup N(a)$ oder $C' = C \cup \{a\} \cup N(b) \cup N(c)$. Man erhält einen Verzweigungsvektor von $(3, 4, 6)$.




*Fall* 4.4: Es gibt keine Kante zwischen den Knoten $a, b, c$ und $|N(a)| = |N(b)| = |N(c)| = 3$ und es gelte $N(a) = \{v, a_1, a_2\}$. Es werden entweder alle Nachbarn von $v$ oder alle Nachbarn von $a$ oder alle Nachbarn von $b, c, a_1, a_2$ in das Vertex Cover übernommen, d.h. $C' = C \cup N(v)$ oder $C' = C \cup N(a)$ oder $C' = C \cup N(b) \cup N(c) \cup N(a_1) \cup N(a_2)$. Man erhält einen Verzweigungsvektor von $(3, 3, 6)$.




*Regel* 5: Alle Knoten haben Grad 4. In diesem Fall wird mit Verzweigungsvektor $(1, 4)$ verzweigt.

## 10.3 Kernbildung mit Linearer Programmierung

Sei $G = (V, E)$ ein Graph. Betrachte das folgende ILP zu Vertex Cover:

$$\min \sum_{x \in V} x_v \tag{10.4}$$

wobei

$$x_v \in \{0, 1\} \qquad \text{für alle } v \in V \tag{10.5}$$

$$x_v + x_u \geq 1 \qquad \text{für alle } \{u, v\} \in E \tag{10.6}$$

Sei $(x_v)_{v\in V}$ eine optimale Lösung des ILPs. Dann ist die Menge $V'$ mit

$$V' := \{v \in V \mid x_v = 1\} \tag{10.7}$$

ein minimales Vertex Cover für $G$.

Im folgenden betrachten wir nun die Relaxierung des ILPs. Hierzu ersetzen wir die Bedingung (10.5) durch die Ungleichungen:

$$x_v \geq 0 \qquad \text{für alle } v \in V \tag{10.8}$$
$$x_v \leq 1 \qquad \text{für alle } v \in V \tag{10.9}$$

**Lemma 10.8.** *Das relaxierte LP zu Vertex Cover hat eine optimale Lösung, für die $x_v \in \{0, \frac{1}{2}, 1\}$ für alle $v \in V$ gilt. Eine solche Lösung kann in polynomieller Laufzeit berechnet werden.*

*Beweis.* Sei $(x_v)_{v\in V}$ eine optimale Lösung der Relaxierung, die die Bedingung des Lemmas nicht erfüllt. Man kann $(x_v)_{v\in V}$ in eine Lösung $(x^*_v)_{v\in V}$ mit folgenden Eigenschaften überführen:

$$\sum_{v\in V} x_v = \sum_{v\in V} x^*_v \tag{10.10}$$
$$x^*_v \in \left\{0, \frac{1}{2}, 1\right\} . \tag{10.11}$$

Sei $\varepsilon := \min\{|x_v|, |x_v - \frac{1}{2}|, |x_v - 1| \ : \ x_v \notin \{0, \frac{1}{2}, 1\}\}$. Zu $v \in V$ definieren wir $x'_v$ und $x''_v$ durch

$$x'_v := \begin{cases} x_v + \varepsilon & \text{falls } 0 < x_v < \frac{1}{2} \\ x_v - \varepsilon & \text{falls } \frac{1}{2} < x_v < 1 \\ x_v & \text{sonst} \end{cases} \tag{10.12}$$

und

$$x''_v := \begin{cases} x_v - \varepsilon & \text{falls } 0 < x_v < \frac{1}{2} \\ x_v + \varepsilon & \text{falls } \frac{1}{2} < x_v < 1 \\ x_v & \text{sonst} \end{cases} \tag{10.13}$$

Dann gilt:

(i) $(x'_v)_{v\in V}$ und $(x''_v)_{v\in V}$ sind wieder Lösungen des LP

(ii) $\sum_{v\in V} x_v = \sum_{v\in V} x'_v = \sum_{v\in V} x''_v$

(iii) mindestens eine der Lösungen $(x'_v)_{v\in V}$ und $(x''_v)_{v\in V}$ enthält weniger Variablen, die nicht in $\{0, \frac{1}{2}, 1\}$ sind als $(x_v)_{v\in V}$:

Zu (i): Sei $\{v, w\} \in E$ und sei o.B.d.A. $x_v \leq x_w$. Falls $x_v = 0$, dann gilt $x_w = x'_w = 1$ und somit gilt auch $x'_v + x'_w \geq 1$. Falls $0 < x_v < \frac{1}{2}$, so gilt $x_w > \frac{1}{2}$ und somit

$$x'_v + x'_w \geq x_v + \varepsilon + x_w - \varepsilon = x_v + x_w \geq 1. \tag{10.14}$$

Für $x_v = \frac{1}{2}$ gilt $x_w \geq \frac{1}{2}$ und damit $x'_v, x'_w \geq \frac{1}{2}$ nach Wahl von $\varepsilon$, also auch $x'_v + x'_w \geq 1$. Ebenso ist die Ungleichung für $x_v > \frac{1}{2}$ erfüllt. Die Argumentation gilt analog für $x''_v$.

Zu (ii): Offenbar gilt

$$\sum_{v\in V} x_v = \frac{1}{2}\left(\sum_{v\in V} x'_v + \sum_{v\in V} x''_v\right) \tag{10.15}$$

nach Defintion von $x'_v$ und $x''_v$ und wegen der Optimalität von $(x_v)_{v\in V}$. Daher gilt

$$\sum_{v\in V} x_v = \sum_{v\in V} x'_v = \sum_{v\in V} x''_v \,. \tag{10.16}$$

Zu (iii): Diese Eigenschaft folgt aus der Wahl von $\varepsilon$.

Nach höchstens $n$ Modifikationen der Form (10.12) bzw. (10.13) ist $x_v \in \{0, \frac{1}{2}, 1\}$ für alle $v \in V$.□

**Lemma 10.9.** *Sei $(x_v)_{v\in V}$ eine Optimallösung mit $x_v \in \{0, \frac{1}{2}, 1\}$ für alle $v \in V$ und zu $r \in \{0, \frac{1}{2}, 1\}$ sei $V_r := \{v \in V \mid x_v = r\}$. Außerdem sei $G_r$ der durch $V_r$ induzierte Teilgraph von $G$. Dann gilt für den Wert vc eines optimalen Vertex Covers*

*(i)* $vc(G_{\frac{1}{2}}) \geq \frac{1}{2}|V_{\frac{1}{2}}|$ *und*

*(ii)* $vc(G_{\frac{1}{2}}) = vc(G) - |V_1|$.

*Beweis.* Zu $W \subseteq V$ und $r \in \{0, \frac{1}{2}, 1\}$ sei $W_r := V_r \cap W$. Dann gelten die folgenden Aussagen:

(a) Falls $S$ ein Vertex Cover von $G$ ist, dann ist $S_r$ Vertex Cover von $G_r$.

(b) Falls $S'$ ein Vertex Cover von $G_{\frac{1}{2}}$ ist, dann ist $S' \cup V_1$ ein Vertex Cover von $G$.

(a) ist offensichtlich. Für jede Kante $\{v, w\}$ mit $v \notin V_{\frac{1}{2}}$ oder $w \notin V_{\frac{1}{2}}$ gilt $v \in V_1$ oder $w \in V_1$ wegen $x_v + x_w \geq 1$. Dann gilt auch (b).

Sei nun $S'$ ein Vertex Cover von $G_{\frac{1}{2}}$ mit $|S'| = vc(G_{\frac{1}{2}})$. Dann folgt

$$|S'| + |V_1| \geq vc(G) \geq \sum_{v\in V} x_v = \frac{1}{2}|V_{\frac{1}{2}}| + |V_1| \tag{10.17}$$

Die erste Ungleichung folgt aus (b), die zweite aus der Optimalität von $(x_v)_{v\in V}$. Also gilt (i). Wegen (b) gilt auch $vc(G_{\frac{1}{2}}) + |V_1| \geq vc(G)$.

Sei nun $S$ ein optimales Vertex Cover von $G$. Wir definieren $(x'_v)_{v\in V}$ durch

$$x'_v := \begin{cases} 1 & \text{falls } v \in S_1 \\ \frac{1}{2} & \text{falls } v \in V_1 \setminus S_1 \text{ oder } v \in V_{\frac{1}{2}} \text{ oder } v \in S_0 \\ 0 & \text{sonst} \end{cases} \tag{10.18}$$

Dann ist $(x'_v)_{v\in V}$ auch eine Lösung des relaxierten LPs. Denn für $\{v, w\} \in E$ gilt: Falls $v, w \in V_1 \cup V_{\frac{1}{2}} \cup S_0$, dann ist $x'_v, x'_w \geq 1$ und somit $x'_v + x'_w \geq 1$. Falls $v \in V_0 \setminus S_0$, dann ist $w \in V_1$, da $(x_v)_{v\in V}$ eine Lösung des relaxierten LPs ist. Außerdem gilt $w \in S$, da $S$ ein Vertex Cover von $G$ ist. Also ist $w \in S_1$ und damit $x'_w = 1$. Es folgt

$$\begin{aligned} \frac{1}{2}|V_{\frac{1}{2}}| + |V_1| &= \sum_{v\in V} x_v && (10.19)\\ &\leq \sum_{v\in V} x'_v && (10.20)\\ &= \frac{1}{2}|S_0| + \frac{1}{2}|V_{\frac{1}{2}}| + \frac{1}{2}|V_1 \setminus S_1| + |S_1| && (10.21)\\ &= \frac{1}{2}|S_0| + \frac{1}{2}|V_{\frac{1}{2}}| + \frac{1}{2}|V_1| + \frac{1}{2}|S_1| && (10.22) \end{aligned}$$

und damit

$$|V_1| \leq |S_0| + |S_1| \tag{10.23}$$

und wegen (a) auch

$$\begin{aligned} vc(G_{\frac{1}{2}}) + |V_1| &\leq |S_{\frac{1}{2}}| + |V_1| && (10.24)\\ &\leq |S_{\frac{1}{2}}| + |S_0| + |S_1| && (10.25)\\ &= |S| && (10.26)\\ &= vc(G) && (10.27) \end{aligned}$$

□

**Folgerung:** Kernbildung zu Instanz $(G, k)$ von Vertex Cover:

(1) Löse LP mit Bedingung $x_v \in \{0, \frac{1}{2}, 1\}$ in polynomieller Zeit.

(2) Setze $k' := k - |V_1|$.

(3) Falls $k' > 0$ und $|V_{\frac{1}{2}}| > 2k'$, dann existiert in $G_{\frac{1}{2}}$ kein Vertex Cover mit $k'$ Knoten und damit auch kein Vertex Cover mit $k$ Knoten in $G$.

(4) Ansonsten gib $(G_{\frac{1}{2}}, k')$ als Kern aus.

Damit kann mit polynomiellem Aufwand ein Kern der Größe maximal $2k'$ ($k' \leq k$) berechnet werden. Für die Berechnung eines Vertex Covers der Größe $k$ in $G$ ergibt sich eine Laufzeit von $\mathcal{O}(p(n) + f(k'))$, wobei $f(k')$ die Laufzeit zur Berechnung eines Vertex Covers der Größe $k'$ in $G_{\frac{1}{2}}$ bezeichne.

# Literaturverzeichnis


[1] B. Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer-Verlag, 1998.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw–Hill, 1990.

[3] J. C. de Pina. *Applications of Shortest Path Methods*. PhD thesis, University of Amsterdam, Netherlands, 1995.

[4] N. Deo. *Graph Theory with Application to Engineering and Computer Science*. Prentice-Hall, Englewood Cliffs, NJ, USA, new edition, 2004.

[5] E. Dijkstra. A note on two problems in connection with graphs. *BIT*, 1:269–271, 1959.

[6] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[7] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[8] A. Frank, T. Ibaraki, and H. Nagamochi. On sparse subgraphs preserving connectivity properties. *J. of Graph Theory*, 17:275–281, 1993.

[9] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1990.

[10] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[11] J. D. Horton. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM Journal on Computing*, 16(2):358–366, 1987.

[12] D. Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI-Wissenschaftsverlag, 1994.

[13] T. Kavitha, K. Mehlhorn, D. Michail, and K. E. Paluch. A faster algorithm for minimum cycle basis of graphs. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 846–857. Springer, 2004.

[14] J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[15] H. Nagamochi and T. Ibaraki. A Linear-Time Algorithm for Finding a Sparse $k$-Connected Spanning Subgraph of a $k$-Connected Graph. *Algorithmica*, 7:583–596, 1992.

[16] R. Niedermeier. *Invitation to Fixed Parameter Algorithms*. Number 31 in Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, 2006.

[17] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. B.I.–Wissenschaftsverlag, 1993.

[18] R. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.

[19] M. Stoer and F. Wagner. A Simple Min Cut Algorithm. In J. Leeuwen, editor, *Second European Symposium on Algorithms, ESA'94*, pages 141–147. Springer-Verlag, Lecture Notes

in Computer Science, vol. 855, 1994.

[20] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983.