Grundlagen der Theoretischen Informatik
Algorithmen und Datenstukturen

Heinrich-Heine-Universität Düsseldorf
Stand: Wintersemester 2006/2007
Dozent: Prof. Dr. Egon Wanke

Inhaltsverzeichnis

Vorbemerkung

Dieses Skript ersetzt kein Lehrbuch. Es kann Ihnen nicht die Teilnahme an den Vorlesungen und Übungen ersparen. Für eine erfolgreiche Teilnahme an den Lehrveranstaltungen müssen Sie die Vorlesungen besuchen, in Lehrbüchern lesen und die wöchentlichen Übungsaufgaben bearbeiten. Sie müssen sich vor allem die Zeit nehmen, den Stoff in aller Ruhe aufzuarbeiten, um die Zusammenhänge wirklich zu verstehen.

Literatur:

- Thomas Ottmann und Peter Widmayer, „Algorithmen und Datenstrukturen“, Spektrum Akademischer Verlag, 4. Auflage, 2002.
- Christoph Meinel, Martin Mundhenk, „Mathrematische Grundlagen der Informatik“, Teubner Verlag, 3. Auflage, 2006.
- Richard Johnsonbaugh, Marcus Schaefer, „Algorithms“, Pearson Verlag, 2004.
- Mark Allen Weiss, „Data Structures and Algorithm Analysis in Java“, Addison Wesley Verlag, 2007.
- Uwe Schöning, „Algorithmik“, Spektrum Akademischer Verlag, 2001.
- Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, „Introduction to Algorithms“, McGraw-Hill, Second Edition, 2001.
- Robert Lafore, „Data Structures & Algorithms in Java“, Second Edition, Sams Publishing, 2003.
- Michael T. Goodrich and Roberto Tamassia, „Data Structures and Algorithms in Java“, John Wiley & Sons, Inc., Second edition, 2001.
- Robert Sedgewick, „Algorithmen“, Addison-Wesley, 2. Auflage, 2002.
- Richard Johnsonbaugh and Marcus Schaefer, „Algorithms“, Pearson Education, First Edition, 2004.
- Herbert Edelsbrunner, „Algorithms in Combinatorial Geometry“, Springer, 1978.
- Franco P. Preparata, „Computational Geometry“, Springer, 1988.
- Rolf Klein, „Algorithmische Geometrie“, Springer Verlag, 2. Auflage. 2005.
- Jon Kleinberg und Eva Tardos, „Algorithm Design“, Addison Wesley, 2006

- M.R. Garey and D.S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, Freeman, 1979
- C.H. Papadimitriou, Computational Complexity, Addision-Wesley, 1994
- J.E. Hopcroft und J.D. Ullman, Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, Addision-Weslay, 1994
- I. Wegener, Theoretische Infomatik, B.G. Teubner, 1993
- T. Nishizeki und N. Chiba. Planar Graphs: Theory and Algorithms, North Holland, 1988
- S. Even, Graph Algorithms, Pitman, 1979
- D. Jungnickel, Graphen, Netzwerke und Algorithmen, BI, 1990
- Algorithms and Data Structures, The Basic Toolbox, Kurt Mehlhorn und Peter Sanders, Springer, 2008
- The Algorithm Design Manual, Steven S. Skiena, Springer, 2008

1 Komplexitätsmaße

Komplexitätsmaße für Algorithmen sind

1. die benötigte Rechenzeit
2. der benötigte Speicherplatz
3. dir Kommunikationsaufwand bei parallelen Prozessen
4. ...

Ziel der Aufwandsmessung ist die Bewertung der Güte von Algorithmen.

Was sind gute und was sind schlechte Algorithmen?

Die beiden wichtigsten Komplexitätsmaße sind die benötigte Rechenzeit und der benötigte Speicherplatz.

Probleme beim Messen der Laufzeit von Algorithmen:

1. Unterschiedlich schnelle Hardware (Computer)
2. Unterschiedlich gute Übersetzer (Compiler)
3. Unterschiedliche Eingabedarstellungen
4. ...

Daher betrachten wir die folgende Abstraktion:

1. Wir zählen die Anzahl der elementaren Schritte:
   (a) arithmetische Operationen
   (b) Vergleiche
   (c) Lese- und Schreiboperationen
   (d) Ein- und Ausgabeoperationen
2. Wir messen die Größe der Eingabe in der Anzahl der elementaren Objekte:
   (a) Anzahl der gegebenen Städte
   (b) Anzahl der gegebenen Zahlen
   (c) Anzahl der Buchstaben in allen gegebenen Namen

Die Rechenzeit ist die Anzahl der benötigten elementaren Schritte in Abhängigkeit von der Größe der Eingabe.

Sei W_n die Menge aller Eingaben der Größe n.
Sei $A_T(w)$ die Anzahl der elementaren Schritte von Algorithmus A für Eingabe w.
Sei $P_n : W_n \to [0,1]$ eine Wahrscheinlichkeitsverteilung für die Eingaben der Größe n, d.h.,

$$\sum_{w \in W_n} P_n(w) = 1.$$

- Die Worst-Case Komplexität (im schlechtesten Fall)

  $$T_A(n) = \sup\{A_T(w) \mid w \in W_n\}$$

  ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus A benötigt, um Eingaben der Größe n zu bearbeiten.

- Die mittlere (erwartete) Komplexität

  $$\overline{T}_A^{P_n}(n) = \sum_{w \in W_n} P_n(w) \cdot A_T(w)$$

  ist die mittlere Anzahl von Schritten, die Algorithmus A benötigt, um Eingaben der Größe n mit Verteilung P_n zu bearbeiten.

1.1 Das O-Kalkül:

Bei der asymptotischen Komplexität werden konstante additive Terme und konstante Faktoren nicht berücksichtigt.

Bemerkung:

1. Bei der Aufwandsabschätzung betrachten wir grundsätzlich nur Funktionen der Art

   $$f : \mathbb{N}_0 \to \mathbb{N}_0,$$

   die für unendlich viele Werte $x \in \mathbb{N}_0$ Funktionswerte $f(x) \neq 0$ annehmen.
   ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$)

2. Der Standardbezeichner für die Eingabegröße ist (meistens) n.

3. Mit Funktionen f, die nicht auf $\mathbb{N}_0$ abbilden, wie zum Beispiel

$$f_1(n) = \log(n),\ f_2(n) = n/10 - 1000,\ f_3(n) = \sqrt{n},$$

verwenden wir bei der Aufwandsabschätzung die Funktionen

$$\overline{f}_1(n) = \lceil \log(n) \rceil,\ \overline{f}_2(n) = \max\{0, \lceil n/10 - 1000 \rceil\},\ \overline{f}_3(n) = \lceil \sqrt{n}\ \rceil.$$

Grundsätzlich betrachten wir bei Aufwandsabschätzungen für eine Funktion $g : \mathbb{R} \to \mathbb{R}$ die Funktion $\overline{g} : \mathbb{N}_0 \to \mathbb{N}_0$ mit $\overline{g}(\max\{0, \lceil n \rceil\}) = \max\{0, \lceil g(n) \rceil\}$, auch wenn wir $g(n)$ für $\overline{g}(\max\{0, \lceil n \rceil\}) = \max\{0, \lceil g(n) \rceil\}$ schreiben.

Definition:

Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion.

Dann ist $O(g) = \left\{ f : \mathbb{N}_0 \to \mathbb{N}_0 \;\middle|\; \begin{array}{l} \exists c\ \in \mathbb{N}_0 : \exists\ x_0 \in \mathbb{N}_0 : \forall\ x \in \mathbb{N}_0 : \\ x \geq x_0 \Rightarrow f(x) \leq c \cdot g(x) \end{array} \right\}$.

„Die Funktionen in $O(g)$ wachsen asymptotisch höchstens so wie g.“

Beispiele:

Sei $g(n) = n$, dann ist

- $7n + 18 \in O(g)$
- $n/2 \in O(g)$
- $4\sqrt{n} \in O(g)$
- $2\log(n) \in O(g)$
- $17 \in O(g)$
- $n^2 \notin O(g)$
- $2^n \notin O(g)$
- $e^{3n} \notin O(g)$

Für jede Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ gilt, wenn $f : \mathbb{N}_0 \to \mathbb{N}_0 \in O(g)$, dann gilt auch:

$$a \cdot f(n) + b \in O(g),\ \forall\, a, b \in \mathbb{R}.$$

Rechenregeln:

$\forall g : \mathbb{N}_0 \to \mathbb{N}_0 : \forall f \in O(g) : \forall c \in \mathbb{R} :$

- $f(n) + c \in O(g)$
- $f(n) \cdot c \in O(g)$

$\forall g_1, g_2 : \mathbb{N}_0 \to \mathbb{N}_0 : \forall f_1 \in O(g_1) : \forall f_2 \in O(g_2) :$

- $f_1 + f_2 \in O(g_1 + g_2)$
- $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$

Oft benutzte (falsche) Schreibweise: $f = O(g)$ anstatt $f \in O(g)$!

Wir schreiben:

$O(n)$	für	$O(g)$,	falls $g(n) = n$
$O(n^2)$	für	$O(g)$,	falls $g(n) = n^2$
$O(\log(n))$	für	$O(g)$,	falls $g(n) = \log(n)$
$O(\sqrt{n})$	für	$O(g)$,	falls $g(n) = \sqrt{n}$
$O(2^n)$	für	$O(g)$,	falls $g(n) = 2^n$
$O(1)$	für	$O(g)$,	falls $g(n) = 1$

Definition:
Ein Algorithmus A hat eine Worst-Case-Laufzeit aus $O(g)$, falls $T_A \in O(g)$.

Beispiel:

Algorithmus: Doppelter Eintrag
Eingabe: $(a_0, \ldots, a_{n-1}) \in \mathbb{N}_0$
Ausgabe: Doppelt $\in \{$true, false$\}$
Doppelt $\Leftrightarrow \exists i, j \in \{0, \ldots, n-1\} : a_i = a_j \wedge i \neq j$

```
(1)   bool Doppelt := false;
(2)   int i := 0;
(3)   int j := 0;
(4)   while (j < n) {
(5)     if ((a[i] = a[j]) and (i ≠ j)) {
(6)       Doppelt := true;
(7)       break; }
(8)     if (i < n-1) {
(9)       i: = i+1; }
(10)    else {
(11)      i := 0;
(12)      j := j+1; } }
```

Im Worst-Case(!) sind alle Zahlen paarweise verschieden. In diesem Fall führt der Algorithmus für Eingabefolgen mit n Zahlen $3+n^2+1+n^2+n\cdot(n-1)+n = 4+3n^2$ Anweisungen aus:

- die drei Anweisungen (1), (2) und (3),
- $n^2 + 1$ mal die Anweisung (4),
- n^2 Mal die Anweisungen (5) und (8/10),
- $n \cdot (n-1)$ Mal die Anweisung (9),
- n Mal die Anweisungen (11) und (12).

$\Longrightarrow$ Algorithmus „Doppelter Eintrag“ hat eine Worst-Case Laufzeit von $O(n^2)$.

Wichtige Aufwandsklassen:

$O(1)$	konstant
$O(\log(n))$	logarithmisch
$O(\sqrt{n})$	Wurzel
$O(n)$	linear
$O(n\log(n))$	
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^c)$	polynomiell (c konstant)
$O(2^n)$	exponentiell
$O(c^n)$	exponentiell ($c > 1$ konstant)

Definition:

Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion.

Dann ist

1. $\Omega(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid g \in O(f)\}$ bzw.

   $$\Omega(g) = \left\{ f : \mathbb{N}_0 \to \mathbb{N}_0 \;\middle|\; \begin{array}{l} \exists\, c \in \mathbb{N}_0 : \exists\, x_0 \in \mathbb{N} : \forall\, x \in \mathbb{N}_0 : \\ x \geq x_0 \Rightarrow c \cdot f(x) \geq g(x) \end{array} \right\}.$$

   $\Omega(g)$ ist die Menge der Funktionen, die asymptotisch mindestens so wachsen wie g.

2. $\Theta(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \in O(g) \wedge g \in O(f)\}$ bzw.

   $$\Theta(g) \quad = \quad \left\{ f : \mathbb{N}_0 \to \mathbb{N}_0 \;\middle|\; \begin{array}{l} \exists\, c_1, c_2 \in \mathbb{N}_0 : \exists\, x_0 \in \mathbb{N}_0 : \forall\, x \in \mathbb{N}_0 : \\ x \geq x_0 \Rightarrow c_1 \cdot f(x) \geq g(x) \wedge c_2 \cdot g(x) \geq f(x) \end{array} \right\}$$

   bzw.

   $$\Theta(g) = O(g) \cap \Omega(g)$$

   $\Theta(g)$ ist die Menge der Funktionen, die asymptotisch genauso wachsen wie g.

Beispiele:

$6n^3 + 2n^2 + 7n - 10$	$\in O(n^3)$	$\in \Omega(n^3)$	$\in \Theta(n^3)$
$n^2 \log(n)$	$\notin O(n^2)$	$\in \Omega(n^2)$	$\notin \Theta(n^2)$
n^k für $k > 0$ und $c > 1$	$\in O(c^n)$	$\notin \Omega(c^n)$	$\notin \Theta(c^n)$
$\log(n)^k$ für $k > 0$	$\in O(n)$	$\notin \Omega(n)$	$\notin \Theta(n)$

Die wichtigsten Aufwandsklassen und ihre Inklusionen:

$$\begin{array}{rcll} O(1) & \subsetneq & O(\log(n)) & \\ & \subsetneq & O(\log(n)^2) & \\ & \subsetneq & O(\sqrt{n}) & \sqrt{n} = n^{\frac{1}{2}} \\ & \subsetneq & O(n) & \\ & \subsetneq & O(n \log(n)) & \\ & \subsetneq & O(n^2) & \\ & \subsetneq & O(n^3) & \\ & \subsetneq & O(2^n) & \\ & \subsetneq & O(3^n) & \end{array}$$

2 Sortieren

Gegeben ist eine Folge von $N > 0$ Elementen $a_0, \ldots, a_{N-1}$; jedes Element a_i hat einen Schlüssel k_i. Gesucht ist eine Permutation π der Zahlen 0 bis $N-1$, so dass die Umordnung der Elemente gemäß der Permutation π die Schlüssel in aufsteigender Reihenfolge bringt:

$$k_{\pi(0)} \leq k_{\pi(1)} \leq \cdots \leq k_{\pi(N-1)}.$$

Eine Permutation der Zahlen 0 bis $N-1$ ist eine bijektive Abbildung

$$\pi : \{0, \ldots, N-1\} \to \{0, \ldots, N-1\}.$$

Diese Problembeschreibung lässt einige Details offen.

Grundsätzliche Voraussetzungen:

1. Die Elemente sind in einem Feld (Array) a auf den Positionen 0 bis $N-1$ ($a[0], \ldots, a[N-1]$) gespeichert; $a[i].k$ ist der Schlüssel und $a[i].s$ der Datensatz auf Position i.

2. Es ist erlaubt zwei Schlüssel zu vergleichen,

   $$a[i].k = a[j].k, \quad a[i].k < a[j].k, \quad a[i].k > a[j].k$$

   Das Ergebnis eines Vergleichs ist die einzige Information, die zur Sortierung herangezogen werden kann.

3. Es ist erlaubt zwei Elemente im Feld zu vertauschen.

2.1 Sortieren durch Auswahl

Methode: Bestimme die Position j_1 an der das Element $a[j_1]$ mit minimalem Schlüssel unter $a[0].k, \ldots, a[N-1].k$ auftritt und vertausche $a[0]$ mit $a[j_1]$. Dann bestimme die Position j_2 an der das Element $a[j_2]$ mit minimalem Schlüssel unter $a[1].k, \ldots, a[N-1].k$ auftritt und vertausche $a[1]$ mit $a[j_2]$, usw. bis alle Elemente am richtigen Platz stehen.

Algorithmische Umsetzung:
Für den Algorithmus verwenden wir eine Mehtode vertausche(Element a [], int i, int j), die das Element an Position i mit dem Element an Position j im Feld a vertauscht.

```
(1)  Auswahl_Sort(Element a [], int N)
(2)  {
(3)    for (int i := 0; i < N-1; i++) {
(4)        int min_i := i;
(5)        for (int j := i+1; j < N; j++){
(6)            if (a[j].k < a[min_i].k)
(7)                min_i := j; }
(8)        vertausche(a, min_i, i); }
(9)  }
```

Beispiel:

Schlüsselfolge
5, 6, 4, 2, 1, 3
1, 6, 4, 2, 5, 3
1, 2, 4, 6, 5, 3
1, 2, 3, 6, 5, 4
1, 2, 3, 4, 5, 6

Analyse: Die Anzahl der Vergleiche ist unabhängig von der anfänglichen Anordnung der Elemente:

$$N-1+N-2+\cdots 1 \;=\; \frac{N(N-1)}{2} \;\in \Theta(N^2)$$

Die Anzahl der Vertauschungen ist unabhängig von der Ausgangsanordnung, da nicht überprüft wird, ob a[min_i] mit sich selbst vertauscht wird.

$$N-1 \;\in \Theta(N).$$

Kann das Verfahren verschnellert werden, indem das Minimum schneller gefunden wird? Nein!

Satz: Jeder Algorithmus zur Bestimmung des Minimums von N Schlüsseln benötigt mindestens $N-1$ Schlüsselvergleiche.

Bemerkung: Ein ausführlicher Beweis ist technisch sehr aufwendig, da alle Voraussetzungen und Randbedingungen für alle denkbaren Modelle vorher genau präzisiert werden müssen.

2.2 Sortieren durch Einfügen

Methode: Die N zu sortierenden Elemente werden nacheinander betrachtet und in die jeweils bereits sortierte, anfangs leere Teilfolge bis zur richtigen Stelle verschoben.
Angenommen wir wollen das i-te Element auf Position $i-1$ in eine bereits sortierte Folge

$$a[0], \ldots, a[i-2]$$

einfügen. Sei r, $0 \leq r \leq i-2$, die größte Position mit $a[r].k \leq a[i-1].k$. Dann vertauschen wir $a[j]$ mit $a[j+1]$ für $j = i-2, \ldots, r+1$.

Algorithmische Umsetzung:

```
(1)  Einfuege_Sort(Element a [], int N)
(2)  {
(3)     for (int i := 1; i < N; i++) {
(4)          int j := i;
(5)          while ((j > 0) and (a[j].k < a[j-1].k)) {
(6)               vertausche(a, j, j-1);
(7)               j := j - 1; } }
(8)  }
```

Beispiel:

Runde	Schlüsselfolge
1	1, 7, 3, 2, 4
2	1, 7, 3, 2, 4
	1, 3, 7, 2, 4
3	1, 3, 2, 7, 4
	1, 2, 3, 7, 4
4	1, 2, 3, 4, 7

Analyse: Im ungünstigsten Fall werden für das Einfügen des i-ten Elementes mit $i = 1, \ldots, N-1$ höchstens $i-1$ Schlüsselvergleiche und $i-1$ Vertauschungen durchgeführt. Daraus folgt, dass insgesamt höchstens

$$1 + 2 + \cdots + N - 1 \;=\; \frac{N(N-1)}{2} \;\in \Theta(N^2)$$

Vergleiche und Vertauschungen durchgeführt werden.

Im günstigsten Fall ist die Folge bereits sortiert. Dann werden $\Theta(N)$ viele Vergleiche und keine Vertauschungen durchgeführt.

Der Aufwand ist abhängig von der Anzahl der Fehlstellungen (Inversionszahl) in der ursprünglichen Anordnung der Elemente.

Im Mittel kann man erwarten, dass die Hälfte der dem i-ten Element vorangehenden Elemente größer als $a[i-1].k$ sind. Die mittlere Anzahl der Vergleiche und Vertauschungen ist somit

$$\sum_{i=1}^{N} \frac{i-1}{2} \;=\; \frac{N(N-1)}{4} \;=\; \Theta(N^2).$$

2.3 Shellsort

Methode: Wähle eine abnehmende und mit 1 endende Folge von Inkrementen

$$h_{t-1}, h_{t-2}, \ldots, h_0,$$

z.B. die Folge $5, 3, 1$.

Betrachte der Reihe nach für jedes h_i mit $i = t-1, \ldots, 0$ alle Teilfolgen mit Elementen im Abstand h_i zueinander.

Genauer: Betrachte für $i = t-1, \ldots, 0$ und $j = 0, 1, \ldots, h_i - 1$ die Folgen $f_{i,j}$ mit den Elementen

$$j,\ j+h_i,\ j+2\cdot h_i,\ j+3\cdot h_i,\ \ldots$$

Sortiere jede Folge $f_{i,j}$ mittels Sortieren durch Einfügen.

Diese auf D.L. Shell zurückgehende Methode nennt man auch
Sortieren mit abnehmenden Inkrementen.

Beispiel:

Sei $t = 3$, $h_2 = 5$, $h_1 = 3$, $h_0 = 1$ und $N = 20$. Dann betrachten wir die Folgen:
$f_{2,0} = (a[0], a[5], a[10], a[15])$
$f_{2,1} = (a[1], a[6], a[11], a[16])$
$f_{2,2} = (a[2], a[7], a[12], a[17])$
$f_{2,3} = (a[3], a[8], a[13], a[18])$

$f_{2,4} = (a[4], a[9], a[14], a[19])$
$f_{1,0} = (a[0], a[3], a[6], a[9], a[12], a[15], a[18])$
$f_{1,1} = (a[1], a[4], a[7], a[10], a[13], a[16], a[19])$
$f_{1,2} = (a[2], a[5], a[8], a[11], a[14], a[17])$
$f_{0,0} = (a[0], \ldots, a[19])$

Die Korrektheit folgt erst aus der Sortierung der letzen Teilfolfge $f_{0,0}$.

Analyse: Für welche Folgen von Inkrementen benötigt das obige Sortierverfahren möglichst wenige Vergleiche bzw. Vertauschungen?

Es gibt eine Reihe überraschender aber insgesamt unvollständiger Antworten.

Man kann zeigen, dass das Verfahren eine Laufzeit von $O(N \cdot \log^2(N))$ hat, wenn als Inkremente alle Zahlen der Form $2^p \cdot 3^q$, $p, q \geq 0$, gewählt werden, die kleiner als N sind.

Beipsiel: Für $N = 5$ sind das die Inkremente

h_i	p	q	$2^p \cdot 3^q$
h_3	2	0	4
h_2	0	1	3
h_1	1	0	2
h_0	0	0	1

Gegeben sei die Schlüsselfolge 5 7 4 9 2 :

Schlüsselfolge	Teilfolgen $f_{i,j}$
5, 7, 4, 9, 2	$f_{3,0} = (5, 2)$, $f_{3,1} = (7)$, $f_{3,2} = (4)$, $f_{3,3} = (9)$
2, 7, 4, 9, 5	$f_{2,0} = (2, 9)$, $f_{2,1} = (7, 5)$, $f_{2,2} = (4)$
2, 5, 4, 9, 7	$f_{1,0} = (2, 4, 7)$, $f_{1,1} = (5, 9)$
2, 5, 4, 9, 7	$f_{0,0} = (2, 5, 4, 9, 7)$
2, 4, 5, 7, 9	–

2.4 Bubblesort

Methode: Durchlaufe das Feld $a[0], \ldots, a[N-1]$ von 0 bis $N-2$ und betrachte je zwei benachbarte Elemente $a[i], a[i+1]$, $0 \leq i < N-1$. Ist $a[i].k > a[i+1].k$, so vertauscht man $a[i]$ und $a[i+1]$. Dieses Durchlaufen wird solange wiederholt, bis

keine Vertauschungen mehr aufgetreten sind. Damit ist das Feld a nach aufsteigenden Schlüsseln sortiert.

Algorithmische Umsetzung:

```
(1)   Bubble_Sort(Element a [], int N)
(2)   {
(3)      bool vertauscht;
(4)      do {
(5)           vertauscht := false;
(6)           for (int i := 0; i < N-1; i++) {
(7)                if (a[i].k > a[i+1].k) {
(8)                     vertausche(a, i, i+1);
(9)                     vertauscht := true; } } }
(10)     while (vertauscht);
(11)  }
```

Beispiel:

Schlüsselfolge
1, 7, 4, 2, 3
1, 4, 7, 2, 4
1, 4, 2, 7, 3
1, 4, 2, 3, 7
1, 2, 4, 3, 7
1, 2, 3, 4, 7

Analyse: Spätestens nach dem ersten Durchlauf ist das größte Element an seinem richtigen Platz. Spätestens nach dem i-ten Durchlauf ist das i-t-größte Element auf seinem richtigen Platz. Spätestens nach dem $(N-1)$-ten Durchlauf ist das Feld a sortiert.

Im günstigsten Fall, wenn das Feld bereits sortiert ist, werden $N-1$ Vergleiche und keine Vertauschungen vorgenommen. Im ungünstigsten Fall, wenn das Feld absteigend sortiert ist, sind $N-1$ Durchläufe nötig. In diesem Fall werden beim i-ten Durchlauf $N-1$ Vergleiche und $N-i$ Vertauschungen durchgeführt. Somit sind im ungünstigsten Fall die Anzahl der Vergleiche und Vertauschungen aus $\Theta(N^2)$.

Man kann zeigen, dass die Anzahl der Vergleiche und Vertauschungen auch im Mittel aus $\Theta(N^2)$ sind.

Bemerkung: Bubblesort ist zwar populär, aber im Grunde schlecht.

Wird das Feld abwechselnd von links nach rechts und umgekehrt durchlaufen, erhält man eine Variante namens Shakersort (schöner Name, aber keine weiteren Vorzüge).

2.5 Quicksort

Quicksort wurde 1962 von C.A.R. Hoare veröffentlicht. Es ist eines der schnellsten internen Sortierverfahren. Es benötigt im schlechtesten Fall $\Theta(N^2)$ viele Vergleichsoperationen, aber im Mittel nur $\Theta(N \cdot \log(N))$ viele Vergleiche.

Quicksort ist wie die andere oben diskutierten Verfahren ein in-situ-Sortierverfahren, es wird kein zusätzlicher Speicherplatz zur Speicherung der Datensätze benötigt (außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen).

Methode: Um eine Folge $F = a[0], \ldots, a[N-1]$ aufsteigend zu sortieren, wählen wir ein beliebiges Element $a[p]$

$$p \in \{0, \ldots, N-1\}$$

zur Aufteilung der Folge F in zwei Teilfolgen F_1 und F_2. Dieses Element nennen wir Pivotelement. Folge F_1 besteht nur aus Elementen von F, dessen Schlüssel kleiner oder gleich $a[p].k$ sind, F_2 nur aus Elementen von F, dessen Schlüssel größer oder gleich $a[p].k$ sind. F_1 und F_2 (falls sie mehr als ein Element enthalten) werden rekursiv mit Quicksort sortiert.

Wesentlich ist der Aufteilungsschritt, d.h., die Wahl des Pivotelementes. Wähle zum Beispiel als Pivotelement das letzte Element der Folge.

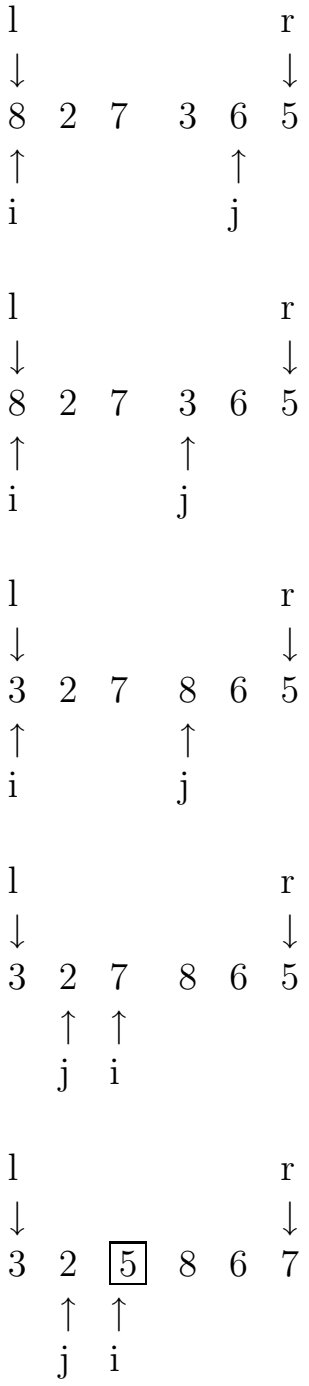
Die folgende Implementierung verändert das Feld so, dass die beiden Teilfolgen F_1 und F_2 nebeneinander stehen.

Algorithmische Umsetzung:

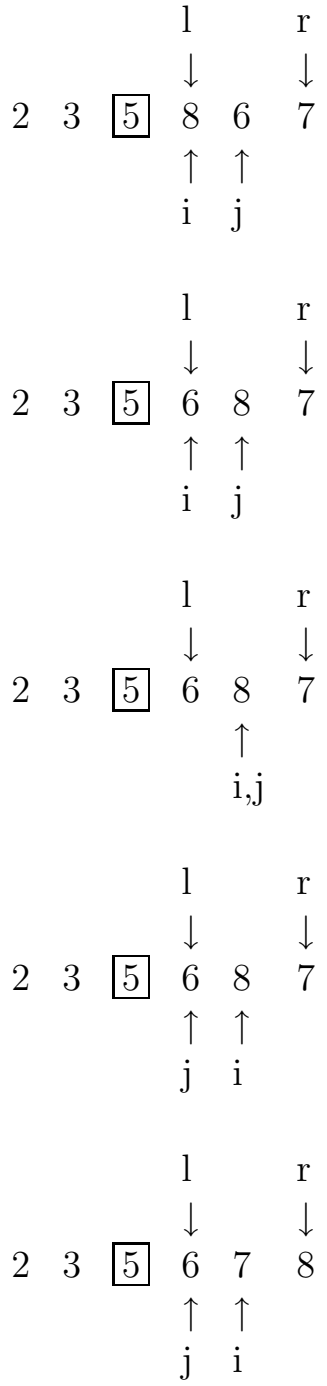
```
(1)  Quick_Sort(Element a [], int l, int r)
(2)  {
(3)     if (l < r) {
```

```
(4)         int p := a[r].k;
(5)         int i := l;
(6)         int j := r-1;
(7)         do {
(8)             while ((a[i].k ≤ p) and (i < r)) i++;
(9)             while ((a[j].k ≥ p) and (j > l)) j −−;
(10)            if (i < j) {
(11)            vertausche(a, i, j); } }
(12)        while (i < j);
(13)        vertausche(a, i, r);
(14)        Quick_Sort(a, l, i-1);
(15)        Quick_Sort(a, i+1, r); }
(16)  }
```

Beispiel:



Die zwei Teilgolgen werden nun rekursiv mit Quicksort sortiert. Bei der linken Teilfolge wird lediglich eine Vertauschung durchgeführt:



Analyse: Angenommen alle Schlüssel sind paarweise verschieden. Dann erfolgen bei jeder Aufteilung von F in F_1, F_2 genau $N + 1$ Vergleiche mit $a[p]$. (Im ungünstigsten Fall wechselt dabei jedes Element einmal seinen Platz.) Die im ungünstigsten Fall insgesamt auszuführende Anzahl von Vergleichen ist somit stark von der Anzahl der Aufteilungsschritte abhängig. Ist das Pivotelement das Elemen mit kleinstem oder größtem Schlüssel, so ist die eine Folge möglicherweise leer und die andere hat nur ein Element (das Pivotelement) weniger als die Ausgangsfolge. Dieser Fall tritt zum Beispiel bei einer bereits aufsteigend sortierten Folge ein. Quicksort benötigt somit im ungünstigsten Fall $\Theta(N^2)$ viele Vergleiche.

Im günstigsten Fall wird die Folge jedesmal genau halbiert. Dadurch ist die Anzahl der Vergleiche bei einer Sortierung von N Elementen

$$C_N \;=\; N+1 \;+\; 2 \cdot C_{(N-1)/2}.$$

Die Lösung obiger Rekurrenzgleichung mit $C_1 = C_0 = 0$ ergibt $\Theta(N \cdot \log(N))$ Vergleiche.

Obwohl die Situation nicht immer so günstig ist, trifft es jedoch zu, dass die Zerlegung im Durchschnitt auf die Mitte fällt.

Die Berücksichtigung der exakten Wahrscheinlichkeit jeder Position der Zerlegung macht die rekurrente Beziehung komplizierter und schwerer auflösbar, doch das Endergebnis ist ähnlich.

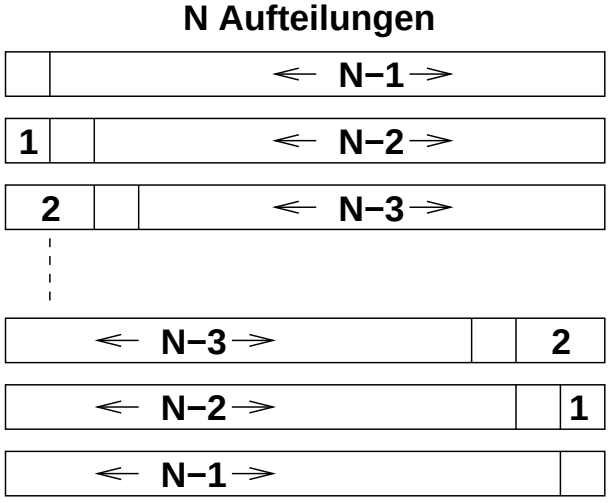
Satz: Quicksort benötigt im Mittel $\Theta(N \cdot \log(N))$ Vergleiche.

Beweis: Die exakte rekurrente Beziehung für die Anzahl der Vergleiche im Mittel lautet

$$C_N \;=\; N+1 \;+\frac{1}{N}\sum_{p=1}^{N}(C_{p-1}+C_{N-p})$$

für $N \geq 2$ und $C_1 = C_0 = 0$.

Der Term $N+1$ ist die Anzahl der Vergleiche zur Teilung. Die Wahrscheinlichkeit, das an Position p zerlegt wird, sei $1/N$. Alle N Positionen sollen gleich wahrscheinlich sein. (Wichtige Voraussetzung!)



Da $C_0+C_1+\cdots+C_{N-2}+C_{N-1} \;=\; C_{N-1}+C_{N-2}+\cdots+C_1+C_0$, kann die Summe wie folgt umgeschrieben werden:

$$C_N \;=\; N+1 \;+2\cdot\frac{1}{N}\cdot\sum_{p=1}^{N} C_{p-1}$$

Multiplikation mit N ergibt

$$N\cdot C_N \;=\; N\cdot(N+1)+2\cdot\sum_{p=1}^{N} C_{p-1}$$

Subtraktion der gleichen Formel für $N-1$ ergibt

$$N\cdot C_N \,-\,(N-1)\cdot C_{N-1} \;=\; N\cdot(N+1)+2\cdot\sum_{p=1}^{N} C_{p-1} \,-\left[\,(N-1)\cdot(N)+2\cdot\sum_{p=1}^{N-1} C_{p-1}\right]$$

bzw.

$$N\cdot C_N \;-(N-1)\cdot C_{N-1} \;=\; N\cdot(N+1) \;-\; (N-1)\cdot N+2\cdot C_{N-1}$$

Durch Addition von $(N-1)\cdot C_{N-1}$ erhalten wir

$$N\cdot C_N \;=\; 2N \;+\; (N+1)\cdot C_{N-1}$$

Division beider Seiten durch $N\cdot(N+1)$ ergibt eine rekurrente Beziehung, die sich wie folgt fortsetzen lässt:

$$\begin{aligned}
\tfrac{C_N}{N+1} &= \tfrac{C_{N-1}}{N}+\tfrac{2}{N+1}\\
&= \tfrac{C_{N-2}}{N-1}+\tfrac{2}{N}+\tfrac{2}{N+1}\\
&= \tfrac{C_{N-3}}{N-2}+\tfrac{2}{N-1}+\tfrac{2}{N}+\tfrac{2}{N+1}\\
&= \tfrac{C_{N-4}}{N-3}+\tfrac{2}{N-2}+\tfrac{2}{N-1}+\tfrac{2}{N}+\tfrac{2}{N+1}\\
&\;\vdots\\
&= \tfrac{C_1}{2}+\tfrac{2}{3}+\ldots+\tfrac{2}{N-2}+\tfrac{2}{N-1}+\tfrac{2}{N}+\tfrac{2}{N+1}
\end{aligned}$$

Durch Umformung und $C_1=0$ erhalten wir

$$\begin{aligned}
\tfrac{C_N}{N+1} &= \tfrac{2}{N+1}-\tfrac{2}{1}-\tfrac{2}{2} \;+\; \tfrac{2}{1}+\tfrac{2}{2}+\tfrac{2}{3}+\ldots+\tfrac{2}{N-1}+\tfrac{2}{N}\\
&= \tfrac{2}{N+1}-3 \;+\; 2\cdot\textstyle\sum_{p=1}^{N}\tfrac{1}{p}\\
&\approx \tfrac{1}{N+1}-3 \;+\; 2\cdot\int_1^N \tfrac{1}{x}dx\\
&= \tfrac{1}{N+1}-3 \;+\; 2\ln N
\end{aligned}$$

Daraus folgt $C_N \in \Theta(N \cdot \log(N))$. $\square$

Quicksort mit beschränkter Rekursionstiefe:

Im ungünstigsten Fall ist die Rekursionstiefe $\Theta(N)$.

Der zusätzlich benötigten Platz für die Rekursionsaufrufe kann gering gehalten werden, wenn das kleinere Teilproblem rekursiv und das größere Teilproblem iterativ direkt gelöst wird.

Dadurch verringert sich die Rekursionstiefe auf $O(\log(N))$.

Algorithmische Umsetzung:

```
(1)   Quick_Sort2(Element a [], int l, int r)
(2)   {
(3)      while (l < r) {
(4)           int k := a[r].k;
(5)           int i := l;
(6)           int j := r-1;
(7)           do {
(8)                while ((a[i].k ≤ k) and (i < r)) i++;
(9)                while ((a[j].k ≥ k) and (j > l)) j−−;
(10)               if (i < j) {
(11)               vertausche(a, i, j); } }
(12)          while (i < j);
(13)          vertausche(a, i, r);
(14)          if ((i-l) < (r-i)) {
(15)               Quick_Sort2(a, l, i-1);    // rekursive Loesung
(16)               l := i+1; }                // iterative Loesung
(17)          else {
(18)               Quick_Sort2(a, i+1, r);    // rekursive Loesung
(19)               r := i-1; } }              // iterative Loesung
(20)  }
```

Verbesserte Pivotwahl:

Die 3-Median-Strategie: Wähle als Pivotelement das mittlere Element von drei Elementen (z.B. von $a[l].k$, $a[r].k$ und $a[\lfloor r + l/2 \rfloor].k$, indem es mit $a[r]$ vertauscht wird.

Die Zufallsstrategie (randomisiertes Quicksort): Wähle als Pivotelement ein zufälliges Element aus $a[l].k, \ldots, a[r].k$ und vertausche es mit $a[r]$.

Der Erwartungswert für die zum Sortieren einer beliebigen aber festen Eingabefolge bei randomisierten Quicksort ist aus $O(N \cdot \log(N))$.

Man nennt ein Sortierverfahren glatt (smooth), wenn es im Mittel N verschiedene Schlüssel in $O(N \cdot \log(N))$ und N gleiche Schlüssel in $O(N)$ Schritten zu sortieren vermag mit einem weichen Übergang zwischen diesen Werten.

Beispiel:

Teile die Folge $a[l], \ldots, a[r]$ in drei Folgen F_l, F_m, F_r auf.

1. F_l enthält die Element mit Schlüssel $< k$.
2. F_m enthält die Element mit Schlüssel $= k$.
3. F_r enthält die Element mit Schlüssel $> k$.

Nach der Aufteilung werden F_l und F_r auf dieselbe Weise sortiert.

Gibt es keine zwei gleiche Schlüssel, so bedeutet dies keine Ersparnis.
Sind alle Schlüssel identisch, erfolgt kein rekursiver Aufruf.

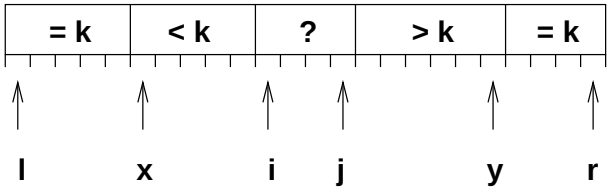
Das oben skizzierte, auf Drei-Wege-Split beruhende Quicksort benötigt im Mittel $O(N \cdot \log(n) + N)$ Schritte, wobei n die Anzahl der verschiedenen Schlüssel unter den N Schlüsseln der Eingabefolge ist.

Algorithmische Umsetzung

```
(1)   Quick_Sort3(Element a [], int l, int r)
(2)   {
(3)     if (l < r) {
(4)         int k := a[r].k;
(5)         int i := l, x := l;
(6)         int j := r-1, y := r-1;
(7)         do {
(8)             while ((a[i].k < k) and (i < r)) i++;
(9)             while ((a[j].k > k) and (j > l)) j−−;
(10)            if (i < j) {
(11)                if ((a[i].k > k) and (a[j].k < k))
(12)                    vertausche(a, i, j);
```

```
(13)                else if ((a[i].k > k) and (a[j].k = k)) {
(14)                    vertausche(a, j, i);      // hänge a[j] links an
(15)                    vertausche(a, i, x);
(16)                    x++; }
(17)                else if ((a[i].k = k) and (a[j].k < k)) {
(18)                    vertausche(a, i, j);      // hänge a[i] rechts an
(19)                    vertausche(a, j, y);
(20)                    y−−;}
(21)                else if ((a[i].k = k) and (a[j].k = k)) {
(22)                    vertausche(a, i, x);      // hänge a[i] links an
(23)                    x++;
(24)                    vertausche(a, j, y);      // hänge a[j] rechts an
(24)                    y−−; } } }
(25)        while (i < j);                        // falls i ≥ j, so ist i die Pivotposition
(26)        i−−;
(27)        x−−;
(28)        for (x ≥ l; i−−; x−−)                 // schiebe alle Pivotelemente von
(29)            vertausche(a, i, x);              // links in die Mitte
(30)        j ++;
(31)        y++;
(32)        for (y ≤ r; j++; y++)                 // schiebe alle Pivotelemente von
(33)            vertausche(a, j, y);              // rechts in die Mitte
(34)        Quick_Sort3(a, l, i);
(35)        Quick_Sort3(a, j, r); }
(36) }
```

Die Pivotelemente werden zuerst am Rand gesammelt (zwischen `l` und `x` sowie zwischen `y` und `r`) und vor dem rekursiven Aufruf in die Mitte transportiert.



Bemerkung: Gibt es auch Sortierverfahren, die im ungünstigsten Fall, also im worst case, mit $O(N \cdot \log(N))$ Vergleichen auskommen? Ja!

2.6 Heapsort

Sortieren mit einer Halde durch geschickte Auswahl.

Methode: Es wird eine Datenstruktur (Heap, Halde) eingesetzt, mit der das Maximum aus einer Menge von N Schlüsseln in einem Schritt bestimmt werden kann.

Definition: Eine Folge $F = a[0], \ldots, a[N-1]$ von N Elementen nennen wir einen Heap, falls $a[i].k \geq a[2 \cdot i+1].k$ und $a[i].k \geq a[2 \cdot i+2].k$ für $i \geq 0$ und $2 \cdot i+1 \leq N-1$ bzw. $2 \cdot i + 2 \leq N - 1$.

Die Elemente $a[2 \cdot i + 1]$ und $a[2 \cdot i + 2]$ sind die Nachfolger von $a[i]$.

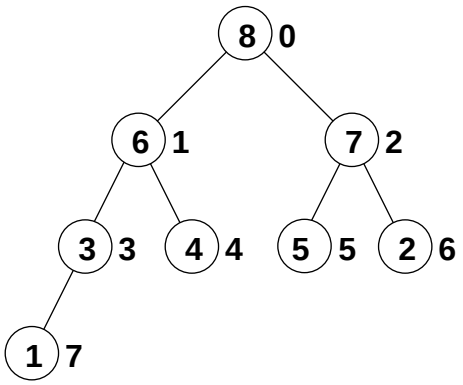
Beispiel: $F = a[0], \ldots, a[7]$ mit $a[0].k = 8$, $a[1].k = 6$, $a[2].k = 7$, $a[3].k = 3$, $a[4].k = 4$, $a[5].k = 5$, $a[6].k = 2$, $a[7].k = 1$ ist ein Heap, weil $8 \geq 6$, $8 \geq 7$, $6 \geq 3$, $6 \geq 4$, $7 \geq 5$, $7 \geq 2$ und $3 \geq 1$.

Folgendes Schaubild erleichtert die Überprüfung der Heapeigenschaft. In den Kreisen stehen die Schlüssel, neben den Kreisen die Positionen in F. Wir gehen jedoch weiterhin davon aus, dass die Elemente in einem Array gespeichert sind.

Felddarstellung:

8	6	7	3	4	5	2	1

Baumdarstellung:



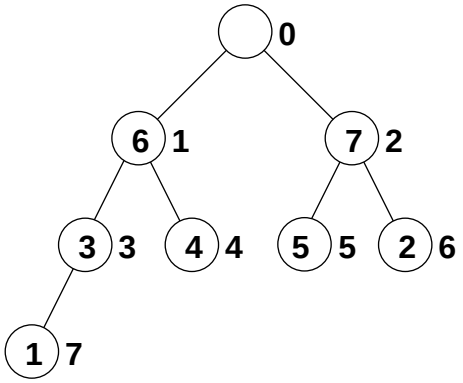
Das Element mit größtem Schlüssel ist $a[0]$.

Das nächst kleinere Element in einem Heap wird bestimmt, indem $a[0]$ aus F entfernt wird, und die Restfolge wieder zu einem Heap transformiert wird.

Bemerkung: Nach dem Entfernen des Elementes mit größtem Schlüssel bleiben zwei Teilheaps zurück.

In unserem Beispiel: $F_1 = a[1], a[3], a[4], a[7]$ und $F_2 = a[2], a[5], a[6]$,

also | - | 6 | - | 3 | 4 | - | - | 1 | und | - | - | 7 | - | - | 5 | 2 | - | .



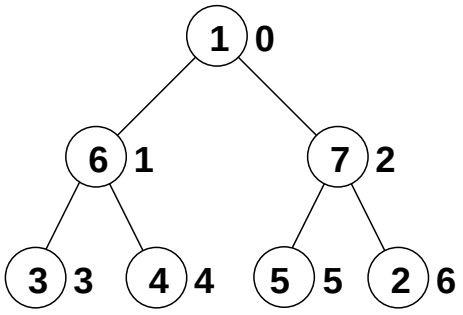
Wir machen daraus wieder einen Heap indem wir zuerst das letzte Element, das Element mit größtem Index, an die erste Position setzen. Dadurch wird jedoch die Heapeigenschaft verletzt.

Trick: Vertausche das erste Element mit dem letzten Element im aktuellen Heap. Das letzte Element gehört anschließend nicht mehr zum neuen Heap, ist aber weiterhin im Feld vorhanden.

Felddarstellung:

1	6	7	3	4	5	2 ‖	8

Baumdarstellung:



Dann lassen wir das neue Element versickern, indem wir es immer wieder mit dem Größeren seiner beiden Nachfolger vertauschen, bis die Schlüssel beider Nachfolger kleiner sind oder das zu versickernde Element unten (an einem Blatt) angekommen ist.

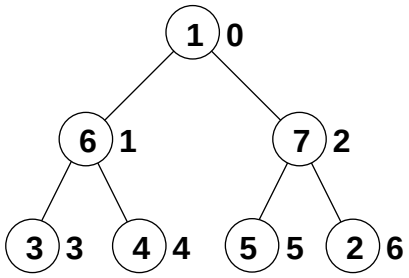
Felddarstellung:

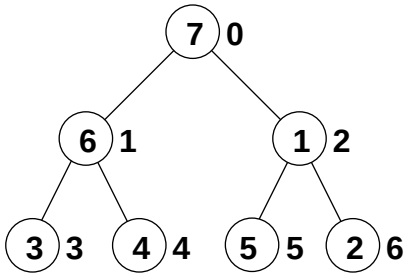
1	6	7	3	4	5	2 ‖	8

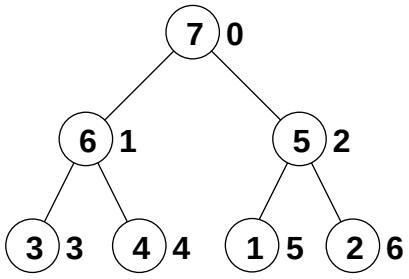
7	6	1	3	4	5	2 ‖	8

7	6	5	3	4	1	2	8

Baumdarstellung:







Algorithmische Umsetzung:

In diesem Algorithmus wird von a[i] bis höchstens a[m] versickert:

```
(1)   versickere(Element a [], int i, int m)
(2)   {
(3)      while (2*i+1 ≤ m) {
(4)           int j := 2*i+1;
(5)           if (2*i+2 ≤ m)
(6)                if (a[2*i+1].k < a[2*i+2].k)
(7)                     j := 2*i+2;
(8)           if (a[i].k < a[j].k) {
(9)                vertausche(a, i, j);
(10)               i := j; }
(11)          else
(12)               i := m; }
```

```
(13)  }
```

Analyse: Es erfolgen $N-1$ viele Vertauschungen außerhalb der Methode versickere( ). Beim Versickern wird ein Element wiederholt mit einem seiner Nachfolger vertauscht. Das Element wandert nach jedem Vertauschen eine Stufe tiefer, wobei sich die Anzahl der Elemente auf jeder Stufe verdoppelt.

Ein Heap mit N Elementen hat $\lceil \log(N+1) \rceil$ viele Stufen. Somit werden beim Versickern höchstens $\lceil \log(N+1) \rceil - 1$ viel Vertauschungen durchgeführt. Dadurch ergibt sich eine obere Schranke von $O(N \cdot \log(N))$ vielen Vergleichen und Vertauschungen.

Die Ausgangsfolge muss jedoch zuerst in einen Heap transformiert werden.

Idee: Wir lassen die Elemente $a[\lfloor \frac{N}{2} \rfloor - 1]$ bis $a[0]$ versickern. Man beginnt also dort zu versickern, wo Elemente als Nachfolger Blätter besitzen. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis letztlich ein einziger Heap übrig bleibt.

Algorithmische Umsetzung:

```
(1)   Heap_Sort(Element a [], int N)
(2)   {
(3)      for (int i := (N/2)-1; i ≥ 0; i−−)
(4)           versickere(a, i, N-1);
(5)      for (int i := N-1; i > 0; i−−) {
(6)           vertausche(a, i, 0);
(7)           versickere(a, 0, i-1); }
(8)   }
```

Analyse: Sei j die Anzahl der Stufen des Heaps mit N Schlüsseln

$$2^{j-1} \leq N \leq 2^j - 1$$

Auf Stufe k, $1 \leq k \leq j$, gibt es höchstens 2^{k-1} Datensätze.

Die Anzahl der Bewegungen und Vergleiche zum Versickern eines Elementes der Stufe k ist proportional zu $j-k$.

Daraus ergibt sich für die Anzahl der Operationen zum Umwandeln einer unsortierten Folge in einen Heap:

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) \;=\; \sum_{k=1}^{j-1} 2^{j-k-1}\cdot k \;=\; 2^{j-1}\sum_{k=1}^{j-1}\frac{k}{2^k} \;<\; N\cdot 2 \in O(N)$$

Erläuterungen:

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) \;=\; \sum_{k=1}^{j-1} 2^{j-k-1}\cdot k$$

k	$2^{k-1}(j-k)$
1	$2^0\cdot(j-1)$
2	$2^1\cdot(j-2)$
3	$2^2\cdot(j-3)$
4	$2^3\cdot(j-4)$
$\dots$	$\dots$
$j-4$	$2^{j-5}\cdot(4)$
$j-3$	$2^{j-4}\cdot(3)$
$j-2$	$2^{j-3}\cdot(2)$
$j-1$	$2^{j-2}\cdot(1)$

k	$2^{j-k-1}\cdot k$
$j-1$	$2^0\cdot(j-1)$
$j-2$	$2^1\cdot(j-2)$
$j-3$	$2^2\cdot(j-3)$
$j-4$	$2^3\cdot(j-4)$
$\dots$	$\dots$
4	$2^{j-5}\cdot(4)$
3	$2^{j-4}\cdot(3)$
2	$2^{j-3}\cdot(2)$
1	$2^{j-2}\cdot(1)$

$$\sum_{k=1}^{j-1}\frac{k}{2^k} \;\le\; 2$$

$$\begin{array}{lcccccccccccccccc}
1 & \ge & \frac{1}{2} & + & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + & \frac{1}{64} & + & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{2} & \ge & & & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + & \frac{1}{64} & + & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{4} & \ge & & & & & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + & \frac{1}{64} & + & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{8} & \ge & & & & & & & \frac{1}{16} & + & \frac{1}{32} & + & \frac{1}{64} & + & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{16} & \ge & & & & & & & & & \frac{1}{32} & + & \frac{1}{64} & + & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{32} & \ge & & & & & & & & & & & \frac{1}{64} & + & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{64} & \ge & & & & & & & & & & & & & \frac{1}{128} & + & \frac{1}{256} & + & \cdots \\
\frac{1}{128} & \ge & & & & & & & & & & & & & & & \frac{1}{256} & + & \cdots \\
\vdots & & & & & & & & & & & & & & & & & & \\
\hline
2 & \ge & \frac{1}{2} & + & \frac{2}{4} & + & \frac{3}{8} & + & \frac{4}{16} & + & \frac{5}{32} & + & \frac{6}{64} & + & \frac{7}{128} & + & \frac{8}{256} & + & \cdots
\end{array}$$

Da der Aufbau eines Heaps aus einer unsortierten Folge in linearer Zeit möglich ist, ist die gesamte Anzahl der Vergleiche und Vertauschungen für Heapsort ebenfalls aus $O(N\cdot\log(N))$.

Bemerkungen:

Eine Vorsortierung der Eingabefolge schadet und nützt der Sortierung nichts.

Heapsort benötigt ebenfalls nur konstant viel zusätzlichen Speicherplatz (in-situ-Sortierverfahren).

2.7 Mergesort

Sortieren durch Mischen.

Mergesort ist eines der ältesten (John von Neumann 1954) Sortierverfahren. Es arbeitet ähnlich wie Quicksort. Die Folge wird jedoch immer in zwei gleich große Teilfolgen aufgeteilt. Die rekursiv sortierten Teilfolgen werden dann in linearer Zeit gemischt.

2-Wege-Mergesort:

Methode: Eine Folge $F = a[0], \ldots, a[N-1]$ von N Elementen wird sortiert, indem sie zunächst in zwei möglichst gleich große Teilfolgen $F_1 = a[0], \ldots, a[\lceil \frac{N}{2} \rceil - 1]$ und $F_2 = a[\lceil \frac{N}{2} \rceil], \ldots, a[N-1]$ aufgeteilt wird.

Dann wird jede dieser Teilfolgen rekursiv mit Mergesort sortiert.

Die Ergebnisse werden gemischt, indem fortlaufend die Elemente mit kleinstem Schlüssel aus den sortierten Folgen entfernt und in die Ergebnisfolge eingefügt wird. Das Element mit kleinstem Schlüssel ist immer entweder das erste Element der sortierten Folge F_1 oder das erste Elemenet der sortierten Folge F_2.

Das Mischen lässt sich im Feld einfach mit Hilfe von zwei Positionsvariablen realisieren.

	F_1	F_2	Ergebnisfolge
	1, 2, 3, 5, 9	4, 6, 7, 8, 10	
	$\uparrow i$	$\uparrow j$	
$1 < 4$	$\uparrow i$	$\uparrow j$	1
$2 < 4$	$\uparrow i$	$\uparrow j$	1,2
$3 < 4$	$\uparrow i$	$\uparrow j$	1,2,3
$5 > 4$	$\uparrow i$	$\uparrow j$	1,2,3,4
$5 < 6$	$\uparrow i$	$\uparrow j$	1,2,3,4,5
$9 > 6$	$\uparrow i$	$\uparrow j$	1,2,3,4,5,6
$9 > 7$	$\uparrow i$	$\uparrow j$	1,2,3,4,5,6,7
$9 > 8$	$\uparrow i$	$\uparrow j$	1,2,3,4,5,6,7,8
$9 < 10$	$\uparrow i$	$\uparrow j$	1,2,3,4,5,6,7,8,9
F_1 leer	$\uparrow i$	$\uparrow j$	1,2,3,4,5,6,7,8,9,10
F_2 leer	$\uparrow i$	$\uparrow j$	1,2,3,4,5,6,7,8,9,10

Wir speichern die sortierte Ergebnisfolge in einem Hilfsarray `b` und kopieren `b` anschließend nach `a` zurück.

Eine mögliche Implementierung:

Für den eigentlichen Algorithmus verwenden wir die Methode
Merge(Element a [], int l, int m, int r), die die Teilfolgen a[l], ...,a[m-1] und a[m], ...,a[r] mischt.

```
(1)   Merge_Sort(Element a [], int l, int r)
(2)   {
(3)     if (l < r) {
(4)         int m := (l+r)/2;       // m ist etwa die Mitte
(5)         Merge_Sort(a, l, m);
(6)         Merge_Sort(a, m+1, r);// beide Teilfolgen sind sortiert
(7)         Merge(a, l, m+1, r); }
(8)   }
```

Analyse: Schlüsselvergleiche werden nur beim Mischen durchgeführt.

Für zwei Teilfolgen mit N_1 und N_2 Elementen werden mindestens $\min(N_1, N_2)$ und höchstens $N_1 + N_2 - 1$ Vergleiche durchgeführt.

Zum Mischen zweier etwa gleich großer Folgen mit N Elementen benötigen wir $\Theta(N)$ viele Vergleiche.

Somit ist die Anzahl der Vergleiche für das Sortieren von N Elementen

$$C_N = C_{\lceil \frac{N}{2} \rceil} + C_{\lfloor \frac{N}{2} \rfloor} + \Theta(N) \in \Theta(N \cdot \log(N)).$$

Das gilt für den schlechtesten ebenso wie für den besten (und damit auch für den mittleren) Fall.

Auch die Anzahl der Vertauschungen (Zuweisungen) ist aus $\Theta(N \cdot \log(N))$.

Das Verfahren ist besonders gut geeignet für den Fall, dass die Elemente in Listen oder auf Bändern gespeichert sind bzw. gespeichert werden können (externe Sortierverfahren).

Reines 2-Wege-Mergesort:

Methode: Eine Folge von N Elementen wird sortiert, indem sortierte Teilfolgen zu immer längere Teilfolgen gemischt werden.

Anfangs wird jedes einzelne Element als eine sortierte Folge betrachtet. Es werden also $(a[0])$ und $(a[1])$ gemischt, $(a[2])$ und $(a[3])$ gemischt, usw.

Danach werden die Teilfolgen mit zwei Elementen zu Teilfolgen mit 4 Elementen gemischt usw.

Beispiel:

2 | 1 | 3 | 9 | 5 | 6 | 7 | 4 | 8 | 10

Nach der ersten Runde:
1 2 | 3 9 | 5 6 | 4 7 | 8 10

Nach der zweiten Runde:
1 2 3 9 | 4 5 6 7 | 8 10

Nach der dritten Runde:
1 2 3 4 5 6 7 9 | 8 10

Nach der vierten Runde:
1 2 3 4 5 6 7 8 9 10

Algorithmische Umsetzung:

```
(1)   Straight_Merge_Sort(Element a [], int l, int r)
(2)   {
(3)      int s := 1;
(4)      while (s < r-l+1) {
(5)           int ll := l;
(6)           while (ll+s < r) {
(7)                int mm := ll+s;
(8)                int rr := mm+s-1;
(9)                if (rr > r)
(10)                    rr := r;
(11)               Merge(a,ll, mm, rr);
(12)               ll := rr+1; }
(13)          s := s*2; }
(14)  }
```

Analyse: Wie 2-Wege-Mergesort.

Natürliches 2-Wege-Mergesort:

Beim natürlichen 2-Wege-Mergesort wird versucht mit möglichst langen bereits sortierten Folgen zu beginnen.

Beispiel:

Die folgende Ausgangsfolge besteht aus vier sortierte Teilfolgen

$2 \mid 1 \quad 3 \quad 9 \mid 5 \quad 6 \quad 7 \mid 4 \quad 8 \quad 10$

Nach der ersten Runde:
$1 \quad 2 \quad 3 \quad 9 \mid 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 10$

Nach der zweiten Runde:
$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10$

Methode: Teile zuerst die Ausgangsfolge $F = a[0], \ldots, a[N-1]$ in längstmögliche, sortierte Teilfolgen. Sortiere dann die Teilfolgen durch wiederholtes Mischen.

Algorithmische Umsetzung:

```
(1)   Natural_Merge_Sort(Element a [], int l, int r)
(2)   {
(3)     bool b := true;
(4)     while (b) {
(5)           b := false;
(6)           int ll := l;
(7)           while (ll < r) {
(8)                 int mm := ll+1;
(9)                 while ((mm ≤ r) and (a[mm-1].k ≤ a[mm].k))
(10)                      mm++;
(11)                int rr := mm+1;
(12)                while ((rr ≤ r ) and (a[rr-1].k ≤ a[rr].k))
(13)                      rr++;
(14)                rr−−;
(15)                if (mm ≤ r) {
(16)                      Merge(a,ll,mm,rr);
(17)                      b := true; }
(18)                ll := rr+1; } }
(19)  }
```

Analyse: Bei der Ermittlung der Anfangsaufteilung werden zusätzlich linear viele Vergleiche durchgeführt. Ansonsten besteht kein Unterschied zum reinen 2-Wege-Mergesort.

Die Anzahl der Vergleiche im schlechtesten Fall ist aus $\Theta(N \cdot \log(N))$.

Der Vorzug des natürlichen 2-Wege-Mergesorts liegt in der Ausnutzung einer Vorsortierung. Im günstigsten Fall ist die Folge bereits sortiert. Dann benötigt natürliches 2-Wege-Mergesort $\Theta(N)$ viele Vergleiche.

Im Mittel gilt für jede Position i, dass beide Ereignisse $a[i].k < a[i+1].k$ und $a[i].k > a[i+1].k$ gleich wahrscheinlich sind, falls alle Schlüssel verschieden sind.

Die Anzahl der Stellen, an denen eine Vorsortierung zu Ende ist, ist somit etwa $N/2$, woraus sich im Mittel etwa $N/2$ Teilfolgen ergeben. Beim reinen 2-Wege-Mergesort erhalten wir nach dem ersten Durchlauf ebenfalls $N/2$ Teilfolgen; daher sparen wir beim natürlichen 2-Wege-Mergesort im Mittel etwa einen Durchlauf.

Die Anzahl der Vertauschungen ist im besten Fall 0 und im Mittel sowie im schlechtesten Fall $\Theta(N \cdot \log(N))$.

2.8 Sortieren durch Fachverteilung

Achtung: Wir wechseln jetzt eine unserer Grundvoraussetzungen!

1. Die Schlüssel sind nun Wörter über einem Alphabet mit genau m-Elementen.
2. Ohne Beschränkung der Allgemeinheit setzen wir im Folgenden voraus, dass alle Schlüssel die gleiche Länge b haben.Falls nötig, fügen wir führende Nullen ein.

Die Schlüssel können als m-adische Zahl aufgefasst werden. Daher nennt man m auch die Wurzel (Radix) der Darstellung.

Sei $z_m(i,k)$ die Ziffer mit Gewicht m^i in Schlüssel k aufgefasst als Zahl zur Basis m.

$$z_{10}(0,517) = 7$$
$$z_{10}(1,517) = 1$$
$$z_{10}(2,517) = 5$$

$$z_2(3,1001) = 1$$

Radix-exchange-sort

Alle Schlüssel sind Binärzahlen ($m = 2$).

Methode: Teile $a[0], \ldots, a[N-1]$ in zwei Teilfolgen. Alle Elemente mit führender 0 im Schlüssel kommen in die linke Teilfolge; alle Elemente mit führender 1 im Schlüssel kommen in die rechte Teilfolge. Die Teilfogen werden rekursiv auf dieselbe Weise sortiert, wobei nun das nächste Bit von links die Rolle des führenden Bits übernimmt.

Die Aufteilung in zwei Teilfolgen kann wie bei Quicksort "in situ" durch Vertauschen von Elementen des Feldes erreicht werden.

Beispiel:

1011	0010	1101	1001	0011	0101	1010
0101	0010	0011 |	1001	1101	1011	1010
0011	0010 |	0101 |	1001	1010	1011 |	1101
0011	0010 |	0101 |	1001 |	1010	1011 |	1101
0010 |	0011 |	0101 |	1001 |	1010 |	1011 |	1101

Analyse: Die Aufteilung benötigt $N+1$ Vergleiche wie bei Quicksort. Die Rekursionstiefe ist immer b (Länge der Schlüssel).

Die Anzahl der Vergleiche ist immer $\Theta(N \cdot b)$.

Ist $b = \log(N)$, dann ist Radix-exchange-sort eine echte Alternative zu Quicksort. Hat man aber nur wenige, verschiedene aber dafür sehr lange Schlüssel, so ist Radix-exchange-sort schlecht.

Radixsort

Methode: Es werden b Verteilungsphasen und Sammelphasen durchlaufen. Für $t = 0, \ldots, b-1$ wird jede Phase einmal durchlaufen.

Verteilungsphase: In der Verteilungsphase werden die Elemente auf m Fächer verteilt. Das i-te Fach F_i nimmt in der $t+1$-ten Verteilungsphase alle Elemente auf, deren Schlüssel an Position t die Ziffer i haben. Die Elemente werden immer oben auf die im Fach bereits vorhandenen Elemente gelegt.

Sammelphase: In der Sammelphase werden die Elemente in den Fächern F_0, $\ldots, F_{m-1}$ gesammelt, indem für $i = m-2, \ldots, 0$ die Elemente im Fach F_{i+1} als Ganzes auf die Elemente im Fach F_i gelegt werden.
In der nachfolgenden Verteilungsphase werden die Elemente von unten nach oben auf die Fächer verteilt.

Beispiel: $m = 10$, $F = 37, 76, 10, 94, 23, 89, 45, 67, 30, 34, 55, 41, 42, 61$

Nach der ersten Verteilung bzgl. der Ziffer auf Position $t = 0$.

30	61			34	55		67		
10	41	42	23	94	45	76	37		89
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Nach der ersten Sammlung:

$$10, 30, 41, 61, 42, 23, 94, 34, 45, 55, 76, 37, 67, 89$$

Nach der zweiten Verteilung bzgl. der Ziffer auf Position $t = 1$.

			37	45					
			34	42		67			
	10	23	30	41	55	61	76	89	94
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Nach der zweiten Sammlung erhalten wir die sortierte Folge:

$$10, 23, 30, 34, 37, 41, 42, 45, 55, 61, 67, 76, 89, 94$$

Korrektheit: Beweis durch Induktion über die betrachtete Position.

Nach der Verteilung und Sammlung bzgl. Position $t = 0$ sind die Elemente bgzl. $t = 0$ sortiert.

Sind die Elemente aufsteigend sortiert, wenn man nur die Positionen $0, \ldots, t-1$ betrachtet, so bleiben sie nach der t-ten Verteilung in jedem Fach aufsteigend sortiert bzgl. der Positionen $0, \ldots, t-1$ sortiert. Nach der t-ten Sammlung sind sie zusätzlich bgzl. Position t sortiert, und somit sortiert bgzl. der Positionen $0, \ldots, t$.

Nach der Verteilung und Sammlung bzgl. Position $t = b-1$ ist die Folge sortiert.

Die Laufzeit von Implementierungen des Radixsort sind aus $\Theta(b \cdot (N+m))$.

Speziafall: Sollen m Elemente mit verschiedenen Schlüsseln im Bereich $0, \ldots, m-1$ sortiert werden, so ist $N = m$ und $b = 1$. Daraus folgt eine Sortierung in linearer Zeit.

2.9 Maße für Vorsortierungen

Beispiele vorsortierter Folgen:

$$\begin{array}{lcccccccccc} F_a & = & 2 & 1 & 4 & 3 & 6 & 5 & 8 & 7 & 9 \\ F_b & = & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 \\ F_c & = & 5 & 1 & 7 & 4 & 9 & 2 & 8 & 3 & 6 \end{array}$$

Die Inversionszahl:

$$\mathrm{inv}(F) = |\{(i,j) \mid 0 \leq i < j \leq N-1 \text{ und } a[i].k > a[j].k\}|$$

Die Inversionszahl liegt immer zwischen 0 für (aufsteigend) sortierte Folgen und $\frac{N(N-1)}{2}$ für absteigend sortierte Folgen.

$$
\begin{array}{rcll}
\mathrm{inv}(F_a) & = & |\{ & (0,1),(2,3),(4,5),(6,7) \qquad\qquad\quad \}| = 4 \\
\mathrm{inv}(F_b) & = & |\{ & (0,4),(0,5),(0,6),(0,7),(0,8) \\
 & & & (1,4),(1,5),(1,6),(1,7),(1,8) \\
 & & & (2,4),(2,5),(2,6),(2,7),(2,8) \\
 & & & (3,4),(3,5),(3,6),(3,7),(3,8) \ \}| = 20 \\
\mathrm{inv}(F_c) & = & |\{ & (0,1),(0,3),(0,5),(0,7),(2,3) \\
 & & & (2,5),(2,7),(2,8),(3,5),(3,7) \\
 & & & (4,5),(4,6),(4,7),(4,8),(6,7) \\
 & & & (6,8) \qquad\qquad\qquad\qquad\quad \}| = 16
\end{array}
$$

Offensichtlich gilt

$$\mathrm{inv}(F) \ = \ |\{(i,j) \mid 0 \le i < j \le N-1 \text{ und } a[i].k > a[j].k\}| \ = \ \sum_{j=2}^{N} h_j$$

mit

$$h_j \ = \ |\{i \mid 0 \le i < j \le N-1 \text{ und } a[i].k > a[j].k\}|$$

h_j ist die Anzahl der Elemente $a[i]$ mit $i < j$, die einen Schlüssel größer als $a[j].k$ haben.

Beim Sortieren durch Einfügen muss das Element $a[j]$ im Abstand h_j vom Ende der bereits sortierten Teilfolge eingefügt werden.

Beispiel:

$$F_c = 5,1,7,4,9,2,8,3,6, \quad \mathrm{inv}(F_c) = 16$$

i	k_i	h_i	
0	5	0	5
1	1	1	1, 5
2	7	0	1, 5, 7
3	4	2	1, 4, 5, 7
4	9	0	1, 4, 5, 7, 9
5	2	4	1, 2, 4, 5, 7, 9
6	8	1	1, 2, 4, 5, 7, 8, 9
7	3	5	1, 2, 3, 4, 5, 7, 8, 9
8	6	3	1, 2, 3, 4, 5, 6, 7, 8, 9

Es gibt Datenstrukturen zur Speicherung sortierter Folgen, die das Suchen und Einfügen eines neuen Elementes im Abstand h vom Ende der Folge in $O(\log(h))$ Schritten erlaubt, ohne das h_i vorher bekannt ist.

Das daraus resultierende Sortierverfahren durch Einfügen hätte eine Laufzeit von

$$O(N + \sum_{j=1}^{N-1} \log(h_j)).$$

Im Worst-Case ist die Laufzeit in $O(N \cdot \log(N))$. Ist die Inversionszahl jedoch linear ($\sum_{j=1}^{N-1} h_j \in O(N)$), so ist die Laufzeit des Sortierverfahrens ebenfalls linear.

Ist dieses Sortierverfahren optimal?

Die Run-Zahl:

$$\text{runs}(F) = |\{i \mid 0 \leq i < N-1 \text{ und } a[i].k > a[i+1].k\}| + 1$$

Die Run-Zahl liegt zwischen 1 für (aufsteigend) sortierte Folgen und N für absteigend sortierte Folgen.

$$\begin{aligned} \text{runs}(F_a) &= 5 \\ \text{runs}(F_b) &= 2 \\ \text{runs}(F_c) &= 5 \end{aligned}$$

Für welches Sortierverfahren könnte dieses Maß von Bedeutung sein?

Die Länge der längsten aufsteigenden Teilfolge (longest ascending subsequence):

$$\text{las}(F) = \max \left\{ t \;\middle|\; \begin{array}{l} \exists i_1, \ldots, i_t \text{ mit} \\ 0 \leq i_1 < i_2 < \cdots i_t \leq N-1 \text{ und} \\ a[i_1].k < a[i_2].k < \cdots a[i_t].k \end{array} \right\}$$

Die Länge der längsten aufsteigenden Teilfolge liegt zwischen N für (aufsteigend) sortierte Folgen und 1 für absteigend sortierte Folgen.

$$\begin{aligned} \text{las}(F_a) &= 5 \\ \text{las}(F_b) &= 5 \\ \text{las}(F_c) &= 4 \end{aligned}$$

Die minimale Anzahl der Elemente, die entfernt werden müssen um eine sortierte Teilfolge zu erhalten:

$$\mathrm{rem}(F) = N - \mathrm{las}(F)$$

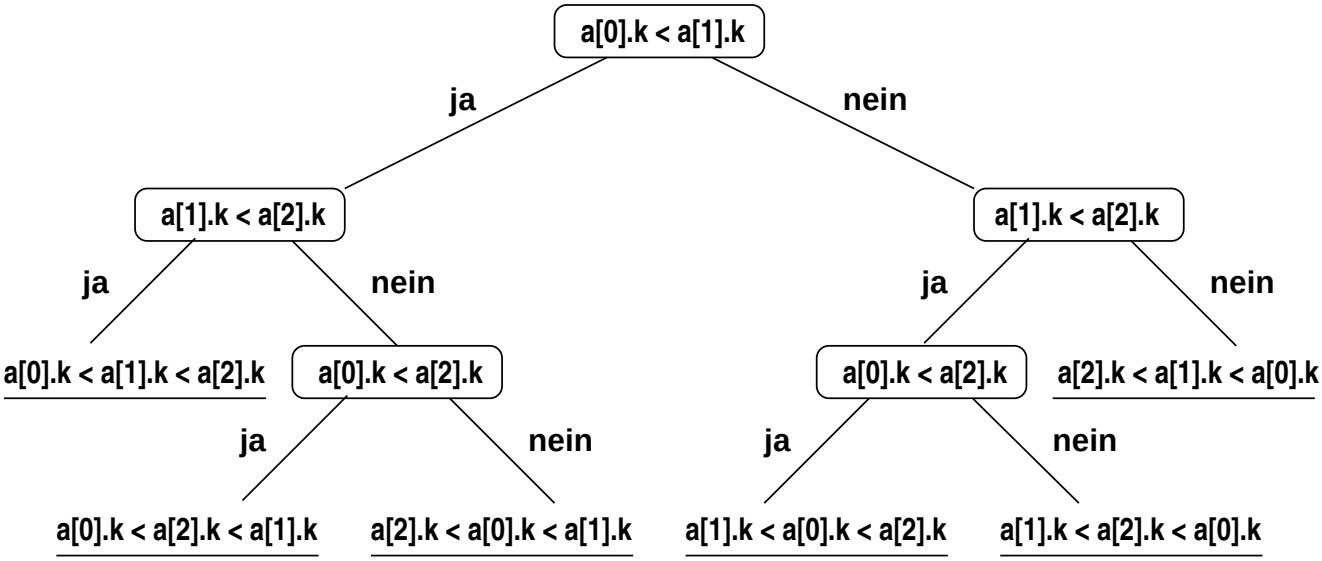
Die Anzahl der Elemente, die entfernt werden müssen um eine sortierte Teilfolge zu erhalten, liegt zwischen 0 für (aufsteigend) sortierte Folgen und $N - 1$ für absteigend sortierte Folgen.

2.10 Untere Schranken

Voraussetzung: Alle Schlüssel sind verschieden.

Jeder Sortieralgorithmus definiert einen Entscheidungsbaum.

Jeder innere Knoten (Entscheidungsknoten) im Entscheidungsbaum repräsentiert einen Schlüsselvergleich. Der linke Sohn repräsentiert den nächsten Schlüsselvergleich bei Antwort "ja", der rechte Sohn repräsentiert den nächsten Schlüsselvergleich bei Antwort "nein". Die Blätter repräsentieren die Elemente der Ausgangsfolge in sortierter Reihenfolge.



Bemerkungen:

- Der Entscheidungsbaum ist ein Binärbaum, d.h., auf Höhe i befinden sich höchstens 2^i Knoten.
- Der Entscheidungsbaum hat $N! = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1$ Blätter, weil es $N!$ viele verschiedene Folgen mit N Elementen gibt.

- Die Anzahl der Entscheidungsknoten auf den Wegen von der Wurzel zu den Blättern entspricht der Anzahl der Vergleiche für das Sortieren der Folgen.
- Der längste Weg von der Wurzel zu einem Blatt enthält mindestens $\lceil \log(N!) \rceil$ Entscheidungsknoten. Daraus folgt, dass jedes Sortierverfahren mindestens $\log(N!) \in \Theta(N \cdot \log(N))$ viele Vergleiche benötigt.

Satz:

$$\log(N!) \in \Theta(N \cdot \log(N))$$

Beweis:

$$\begin{aligned}
\left(\left\lfloor \tfrac{N}{2} \right\rfloor + 1\right)^{\lceil \frac{N}{2} \rceil} &= \overbrace{\left\lfloor \tfrac{N}{2} \right\rfloor + 1 \cdot \left\lfloor \tfrac{N}{2} \right\rfloor + 1 \cdot \ldots \cdot \left\lfloor \tfrac{N}{2} \right\rfloor + 1 \cdot \left\lfloor \tfrac{N}{2} \right\rfloor + 1}^{\lceil \frac{N}{2} \rceil} \\
N! &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot \left\lfloor \tfrac{N}{2} \right\rfloor + 1 \cdot \left\lfloor \tfrac{N}{2} \right\rfloor + 2 \cdot \ldots \cdot N-1 \cdot N \\
N^N &= \underbrace{N \cdot N \cdot N \cdot \ldots \cdot N \cdot N \cdot N \cdot \ldots \cdot N \cdot N}_{N}
\end{aligned}$$

$$\begin{aligned}
\left(\left\lfloor \tfrac{N}{2} \right\rfloor + 1\right)^{\lceil \frac{N}{2} \rceil} &\leq N! \leq N^N \\
\log\left(\left(\left\lfloor \tfrac{N}{2} \right\rfloor + 1\right)^{\lceil \frac{N}{2} \rceil}\right) &\leq \log(N!) \leq \log\left(N^N\right) \\
\left(\left\lceil \tfrac{N}{2} \right\rceil\right) \cdot \log\left(\left\lfloor \tfrac{N}{2} \right\rfloor + 1\right) &\leq \log(N!) \leq N \cdot \log(N)
\end{aligned}$$

Da $\log(\frac{N}{2}) = \log(N) - 1$, folgt $\log(N!) \in \Theta(N \log(N))$.
□

Ein Sortierverfahren nutzt eine Vorsortierung optimal aus, wenn im Entscheidungsbaum die Blätter der maximal vorsortierten Folgen so nah wie möglich bei der Wurzel sind.

Sei $m(F)$ ein Maß für die Vorsortierung einer Folge F.

Für eine Folge F sei $r_{m,F}$ die Anzahl aller Folgen F' gleicher Länge, die genau so gut vorsortiert sind wie F.

$$r_{m,F} \;=\; |\{F' \mid m(F') \leq m(F)\}|$$

Da der Entscheidungsbaum ein Binärbaum ist, gibt es mindestens eine Folge F_0 mit $m(F_0) \leq m(F)$, deren Abstand von der Wurzel mindestens $\lceil \log(r_{m,F}) \rceil$ ist.

Da jedoch immer mindestens $N-1$ Vergleiche notwendig sind, ist $\Omega(N+\log(r_{m,F}))$ eine untere Schranke für die Anzahl der Vergleiche zum Sortieren vorsortierter Folgen.

3 Suchen

3.1 Das Auswahlproblem

Voraussetzung: Alle Schlüssel sind verschieden.

Gesucht ist das Element mit dem i-t kleinsten Schlüssel in einer Folge $a_0, \cdots, a_{N-1}$ bzw. der Median der Folge.

Der Median einer Folge $a_0, \cdots, a_{N-1}$ mit paarweise verschiedenen Schlüsseln ist das Element a_i, für dass $\lfloor \frac{N}{2} \rfloor$ Elemente der Folge einen Schlüssel kleiner als k_i haben und $\lceil \frac{N}{2} \rceil - 1$ Elemente der Folge einen Schlüssel größer als k_i haben.

Beispiele:

$N = 9$, $\lfloor \frac{N}{2} \rfloor = 4$ kleinere Schlüssel, $\lceil \frac{N}{2} \rceil - 1 = 4$ größere Schlüssel
$N = 10$, $\lfloor \frac{N}{2} \rfloor = 5$ kleinere Schlüssel, $\lceil \frac{N}{2} \rceil - 1 = 4$ größere Schlüssel

Es gilt immer

$$\left\lfloor \frac{N}{2} \right\rfloor \quad \geq \quad \left\lceil \frac{N}{2} \right\rceil - 1$$

Methode 1: Suche in der Folge i-mal das Element mit kleinstem Schlüssel und entferne es.

Laufzeit: $\Theta(i \cdot N)$

Der Median kann so in Zeit $\Theta(N^2)$ gefunden werden.

Methode 2: Sortiere die Folge und suche das Element mit dem i-t kleinsten Schlüssel.

Laufzeit: $O(N \cdot \log(N))$

Der Median kann so in Zeit $O(N \cdot \log(N))$ gefunden werden.

Methode 3: (Mit einem Heap) Erstelle aus den gegebenen N Elementen einen Heap mit umgekehrter Ordnung, d.h., die Schlüssel der Nachfolger sind stets größer als die Schlüssel der Vorgänger.

Entferne dann i mal das kleinste Element.

Laufzeit: $O(N + i \cdot \log(N))$

Der Median kann so ebenfalls in Zeit $O(N \cdot \log(N))$ gefunden werden.

Methode 4: Teile die Folge wie bei Quicksort bezüglich eines geeignet gewählten Pivotelementes in zwei Teilfolgen F_1, F_2 auf. F_1 enthält nur Elemente mit Schlüssel, die kleiner als das Pivotelement sind; F_2 enthält nur Elemente mit Schlüssel, die größer als das Pivotelement sind.

- Falls F_1 genau $i-1$ Elemente enthält, so ist der i-t kleinste Schlüssel das Pivotelement.
- Falls F_1 mehr als $i-1$ Elemente enthält, so kommt der i-t kleinste Schlüssel in F_1 vor, ansonsten ist der i-kleinste Schlüssel in F_2.

Suche dann den i-t kleinsten Schlüssel in F_1 bzw. den $i-(|F_1|+1)$ kleinsten Schlüssel in F_2.

Kann das Pivotelement so gewählt werden, dass $q \cdot N$ Elemente in der einen Teilfolge und $(1-q)N$ Elemente in der anderen Teilfolge sind, für einen konstanten Bruchteil $q \geq 1/2$, so gilt offenbar folgende Gleichung:

Sei $T(N)$ die Anzahl der Schritte des Verfahrens für das Finden des Elementes mit i-t kleinstem Schlüssels in einer Folge von N Elementen.

Dann ist

$$\begin{aligned} T(N) &= & & & T(q\cdot N) & +c\cdot N \\ &= & & T(q^2\cdot N) & +q\cdot c\cdot N & +c\cdot N \\ &= & T(q^3\cdot N) & +q^2\cdot c\cdot N & +q\cdot c\cdot N & +c\cdot N \\ &= T(q^4\cdot N) & +q^3\cdot c\cdot N & +q^2\cdot c\cdot N & +q\cdot c\cdot N & +c\cdot N \end{aligned}$$

für eine entsprechende Konstante c und somit

$$T(N) \;\leq\; c\cdot N\sum_{i=0}^{\infty} q^i \;=\; c\cdot N\cdot\frac{1}{1-q} \;\in O(N).$$

Unter diesen Voraussetzungen kann das i-t kleinste Element in linearer Zeit gefunden werden.

Problem: Finde ein geeignetes Pivotelement.

Lösung:

Um in einer Folge von N Elementen das i-t kleinste Element zu finden, benutzt man zum Beispiel die „Median der Mediane Strategie“ von Blum, Floyd, Pratt, Rivest, Tarjan (1972).

3.2 Die „Median der Mediane Strategie“

1. Falls N kleiner als eine gewisse Konstante C ist, berechne das i-t kleinste Element direkt. (C wird später bestimmt.)
2. Ansonsten teile die Folge mit N Elementen in $\lfloor \frac{N}{5} \rfloor$ Gruppen zu je fünf Elementen und höchstens eine Gruppe mit < 5 Elementen auf.
3. Bestimme den Median jeder Gruppe direkt.

   Man erhält so $\lceil \frac{N}{5} \rceil$ Mediane in Zeit $O(N)$.
4. Suche mit dem gleichen Verfahren rekursiv den Median v der Mediane.
5. Wähle den Median v der Mediane als Pivotelement und teile die Folge in zwei Teilfolgen F_1, F_2 auf.

   F_1 enthält k Elemente mit Schlüssel, die kleiner als v sind; F_2 enthält $N-k-1$ Elemente mit Schlüssel, die größer als v sind.

   Die Aufteilung kann in Zeit $O(N)$ erfolgen.
6. Falls $i \leq k$ ist, suche rekursiv das i-t kleinste Elemente in F_1, falls $i > k+1$ ist suche rekursiv das $i-(k+1)$-t kleinste Element in F_2, falls $i = k+1$ ist, hat man das i-t kleinste Element der Ausgangsfolge gefunden (nämlich v).

Beispiel: Gegeben sei die Folge

25 12 31 4 19 35 11 2 14 22 18 27 9 1 33 13 23 8 3 6 34 10 21 16 7

26 32 30 24 20 29 15 17 28.

Gesucht wird der 5-t kleinster Schlüssel. Teile die Folge in Gruppen ein und bestimme den Median:

$\underbrace{25\ 12\ 31\ 4\ 19}_{\text{Median 19}}$ $\underbrace{35\ 11\ 2\ 14\ 22}_{\text{Median 14}}$ $\underbrace{18\ 27\ 9\ 1\ 33}_{\text{Median 18}}$ $\underbrace{13\ 23\ 8\ 3\ 6}_{\text{Median 8}}$ $\underbrace{34\ 10\ 21\ 16\ 7}_{\text{Median 16}}$

$\underbrace{26\ 32\ 30\ 24\ 20}_{\text{Median 26}}$ $\underbrace{29\ 15\ 17\ 28}_{\text{Median 28}}$

Der Median der Mediane ist $v = 18$.

Teile nun die Folge in zwei Teilfolgen F_1 und F_2 mit $v = 18$ als Pivotelement:

12 4 11 2 14 9 1 13 8 3 6 10 16 7 15 17 $|18|$ 25 31 19 35 22

27 33 34 21 26 32 30 24 20 23 29 28

F_1 enthält 13 Elemente mit Schlüsseln, die kleiner als 18 sind. Suche also rekursiv den 5-t kleinsten Schlüssel in F_1:

$$\underbrace{12\ 4\ 11\ 2\ 14}_{\text{Median 11}}\ \underbrace{9\ 1\ 13\ 8\ 3}_{\text{Median 8}}\ \underbrace{6\ 10\ 16\ 7\ 15}_{\text{Median 10}}\ \underbrace{17}_{\text{Median 17}}$$

Der Median der Mediane ist $v = 11$.

Teile nun die Folge F_1 in zwei Teilfolgen $\overline{F_1}$ und $\overline{F_2}$ mit $v = 11$ als Pivotelement:

$$4\ 2\ 9\ 1\ 8\ 3\ 6\ 10\ 7\ |11|\ 12\ 14\ 13\ 16\ 15\ 17$$

$\overline{F_1}$ enthält 9 Elemente mit Schlüsseln, die kleiner als 11 sind. Suche rekursiv den 5-t kleinsten Schlüssel in $\overline{F_1}$. Hier wäre es sinnvoll, den 5-t kleinsten Schlüssel direkt zu bestimmen. Somit ergibt sich 6 als der 5-t kleinste Schlüssel der Gesamtfolge.

Analyse:

Anzahl der Elemente für den rekursiven Aufruf der Methode in Schritt 6:
Jede Gruppe (außer die Gruppe, die den Median der Mediane v enthält, und die Gruppe mit weniger als 5 Elementen) mit einem Median kleiner als v enthält mindestens drei Elemente mit Schlüssel kleiner als v.

In der Ausgangsfolge gibt es genau $\left\lfloor \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rfloor$ viele Gruppen mit einem Median kleiner als v und somit mindestens $\left\lfloor \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rfloor - 1$ Gruppen mit genau 5 Elementen und einen Median kleiner als v und somit mindestens

$$3 \cdot \left(\left\lfloor \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rfloor - 1 \right) \ \geq \ 3 \cdot \left(\left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 2 \right) \ \geq \ \frac{3N}{10} - 6$$

Elemente mit Schlüssel kleiner als v.

Bemerkung: $\lfloor a \rfloor \geq \lceil a \rceil - 1$

Jede Gruppe (außer die Gruppe, die v enthält, und die Gruppe mit weniger als 5 Elementen) mit einem Median größer als v enthält mindestens drei Elemente mit Schlüssel größer als v.

In der Ausgangsfolge gibt es $\left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 1$ viele Gruppen mit einem Median größer als v und somit mindestens $\left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 2$ Gruppen mit genau 5 Elementen und einen Median größer als v und somit mindestens

$$3 \cdot (\left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 2) \geq \frac{3N}{10} - 6$$

Elemente mit Schlüssel größer als v.

Anweisung 6 des Verfahrens muss somit für höchstens $N - (\frac{3N}{10} - 6) = \frac{7N}{10} + 6$ Elemente aufgerufen werden.

Laufzeitanalyse:

Sei $T(N)$ die Anzahl der Schritte des Verfahrens für das Finden des i-t kleinsten Elementes in einer Folge von N Elementen.

Dann ist

$$T(N) \leq T\left(\left\lceil \frac{N}{5} \right\rceil\right) + T\left(\left\lceil \frac{7}{10}N + 6 \right\rceil\right) + a \cdot N$$

für eine Konstante a.

Dabei ist

$$T\left(\left\lceil \frac{N}{5} \right\rceil\right)$$

ist die Anzahl der Schritte für den rekursiven Aufruf in Anweisung 4,

$$T\left(\left\lceil \frac{7}{10}N + 6 \right\rceil\right)$$

ist die Anzahl der Schritte für den rekursiven Aufruf in Anweisung 6, und $a \cdot N$ der Aufwand in den Schritten (1), (2), (3) und (5).

Wir behaupten, dass $T(N) \leq c \cdot N$ ist für eine Konstante c, dass heisst $T(N) \in O(N)$.

Voraussetzung ist jedoch, dass die Argumente für T auf der rechten Seite der Ungleichung alle kleiner als N sind, ansonsten terminiert das Verfahren nicht. Dies ist sicher der Fall für $(7/10)N + 6 < N$ also $N > 20$.

Induktionsbeweis:

Angenommen $T(M) \leq c \cdot M$ sei bereits bewiesen für alle Argumente M kleiner als N und größer als 20.

Dann ist

$$\begin{aligned} T(N) &\leq c \cdot \lceil \tfrac{N}{5} \rceil & + c \cdot \lceil (\tfrac{7}{10}N + 6) \rceil & & + a \cdot N \\ &\leq c \left(\cdot \tfrac{1}{5}N + 1 \right) & + c \cdot \left(\tfrac{7}{10}N + 6 + 1 \right) & & + a \cdot N \\ &\leq c \cdot \tfrac{1}{5}N + c & + c \cdot \tfrac{7}{10}N + 6c + c & & + a \cdot N \\ &= \tfrac{9}{10}cN + 8c & & & + a \cdot N \end{aligned}$$

Wählt man zum Beispiel $c = 100a$, so ist der letzte Ausdruck kleiner oder gleich $c \cdot N$ für alle $N \geq 89$, wie folgende Rechnung zeigt.

$$\begin{aligned} \tfrac{9}{10}cN + 8c + aN &\leq cN \\ \tfrac{900}{10}aN + 800a + aN &\leq 100aN \\ 91N + 800 &\leq 100N \\ 800 &\leq 9N \\ 88.88\cdots &\leq N \end{aligned}$$

Da wir das i-t kleinste Element in einer Folge mit höchstens 89 Elementen in konstanter Zeit finden können, gilt insgesamt die folgende Aussage:

Satz: Das i-t kleinste Element in einer Folge von N Elementen kann in $O(N)$ vielen Schritten gefunden werden.

Bemerkungen:

- Die Aussage ist von großer prinzipieller Bedeutung!
- Praktisch lohnt sich diese Methode erst bei sehr großen Datenmengen.
- Wird bei Quicksort der Median als Pivotelement gewählt, ist die Anzahl der Vergleiche im besten, im mittleren und im schlechtesten Fall in $\Theta(N \cdot \log(N))$.

3.3 Suchen in linearen Listen

Gesucht wird ein Element mit Schlüssel k.

Lineares Durchsuchen einer Liste mit N Elementen benötigt N Vergleiche im ungünstigsten Fall, und $N/2$ Vergleiche im Mittel.

Es kann schneller gesucht werden, wenn die Folge bereits sortiert ist.

Binäre Suche

Methode:

1. Falls $N = 0$, breche die Suche ab.
2. Vergleiche den Schlüssel des Elementes auf Position $\lceil \frac{N-1}{2} \rceil$ mit dem Schlüssel k.

   Ist $a[\lceil \frac{N-1}{2} \rceil].k < k$, dann durchsuche rekursiv die Folge $a[\lceil \frac{N-1}{2} \rceil + 1], \ldots, a[N-1]$.

   Ist $a[\lceil \frac{N-1}{2} \rceil].k > k$ dann durchsuche rekursiv die Folge $a[0], \ldots, a[\lceil \frac{N-1}{2} \rceil - 1]$.

   Ist $a[\lceil \frac{N}{2} \rceil].k = k$ dann ist das Element mit Schlüssel k gefunden.

Die Methode kann rekursiv als auch iterativ implementiert werden.

Beispiel:
Gegeben Sei die bereits sortierte Folge im Feld a:

2	3	5	9	14	20	22	23

Wir suchen den Schlüssel $k = 5$ in der Folge. Wir vergleichen im ersten Schritt (nach der Mehtode) den Schlüssel auf Position $a[4]$ mit dem Schlüssel k. Da $14 > 5$, suchen wir in der linken Teilfolge rekursiv weiter:

2	3	5	9

Wir vergleichen den Schlüssel auf Position $a[2]$ mit dem Schlüssel k und haben den Schlüssel gefunden.

Analyse: Binäres Suchen benötigt höchstens $O(\log(N))$ Vergleiche.
Im Mittel benötigt binäres Suchen ebenfalls $O(\log(N))$ Vergleiche.

Fibonacci-Suche

Die Division zur Bestimmung der mittleren Position kann vermieden werden.

Methode: Dabei wird die Aufeilung der Folge entsprechend der Fibonacci-Zahlen vorgenommen.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ &\vdots \\ F_n &= F_{n-1} + F_{n-2} \text{ für } (n \geq 2) \end{aligned}$$

Angenommen das zu durchsuchende Feld hat die Länge $N = F_n - 1$ für ein $n > 0$. Dann teilen wir die Folge auf in die ersten $F_{n-2} - 1$ Elemente, das Element auf Position F_{n-2} und die letzten $F_{n-1} - 1$ Elemente.

$$\begin{array}{lcccccl} F_n & = & F_{n-2} & & & + & F_{n-1} \\ F_n - 1 & = & F_{n-2} - 1 & + & 1 & + & F_{n-1} - 1 \end{array}$$

Um für einen Suchbereich der Länge $F_n - 1$ die Aufteilungsposition F_{n-2} zu finden, merken wir uns die beiden Fibonacci-Zahlen $f = F_n$ und $g = F_{n-1}$.

Angenommen für den zu durchsuchenden Bereich speichern wir das Paar (f, g), dann wird k mit $a[(f - g) - 1].k$ verglichen:

- Falls $a[(f - g) - 1].k < k$, dann suchen wir rechts weiter.

  Der rechte Bereich hat die Länge $g - 1$.

  Als neues Paar für den rechten Bereich wählen wir $(f', g') = (g, f - g)$.

- Falls $a[(f - g) - 1].k > k$, dann suchen wir links weiter.

  Der linke Bereich hat die Länge $f - g - 1$.

  Als neues Paar für den linken Bereich wählen wir $(f', g') = (f - g, g - (f - g))$.

Beispiel: Bei einer Folge mit $F_7 - 1 = 12$ Elementen teilen wir wie folgt auf:

- die linke Teilfolge mit $F_5 - 1 = 4$ Elementen,
- das Element auf Position $a[F_5 - 1] = a[4]$ (dessen Schlüssel mit dem Schlüssel k verglichen wird),
- die rechte Teilfolge mit $F_6 - 1 = 7$ Elementen.

Gesucht sei der Schlüssel $k = 3$ in der Folge

2	3	6	9	12	16	18	19	23	26	34	36

Der Schlüssel $k = 3$ wird mit dem Schlüssel auf Position $a[4]$ verglichen: Da $a[4] > k$, suchen wir rekursiv in $a[0], \cdots, a[3]$ weiter.

Die Folge $a[0], \cdots, a[3]$ mit $F_5 - 1 = 4$ Elementen teilen wir wieder auf:

- eine linke Teilfolge mit $F_3 - 1 = 1$ Elementen,
- das Element auf Position $a[F_3 - 1] = a[1]$ (dessen Schlüssel mit dem Schlüssel k verglichen wird),
- die rechte Teilfolge mit $F_4 - 1 = 2$ Elementen.

Da $a[1] = k$, wird die Suche erfolgreich beendet.

Analyse: Wieviele Schlüsselvergleiche werden bei der Suche nach einem Schlüssel k maximal ausgeführt?

Bei einem Suchbereich der Länge $F_j - 1$ ist die Länge des nächstens Suchbereichs höchstens $F_{j-1} - 1$. Also sind für die Suche in einem Bereich der Länge $F_n - 1$ höchstens n Schlüsselvergleiche erforderlich.

Nun ist

$$F_n \approx c \cdot 1.618^n$$

mit einer Konstanten c.

Genauer:

Die maximal erforderliche Anzahl von Vergleichen ist somit

$$O(\log_{1.618}(N+1)) \;=\; O\left(\log_{1.618}(2) \cdot \log(N)\right) \;=\; O(\log(N)).$$

Die im Mittel ausgeführte Anzahl von Vergleichen ist ebenfalls in $O(\log(N))$.

Bemerkung: Seien $a, b > 1$, dann gilt:

$$\log_b(N) \quad = \quad \log_b(a) \cdot \log_a(N)$$

weil:

$$b^{\log_b(N)} \;=\; N \;=\; a^{\log_a(N)} \;=\; \left(b^{\log_b(a)}\right)^{\log_a(N)} \;=\; b^{\log_b(a) \cdot \log_a(N)}$$

Exponentielle Suche:

Ist die Länge der Folge vor Beginn der Suche nicht bekannt, so eignet sich die exponentielle Suche.

Methode: Wir bestimmen zunächst in exponentiell wachsenden Schritten einen Bereich, in dem der Schlüssel k liegen muss.

```
(1)   i := 1;
(2)   while (k > a[i-1].k)
(3)         i:=i + i;
```

Danach gilt

$$a[i/2].k \leq k \leq a[i-1].k.$$

Setzen wir voraus, dass die Folge aufsteigend sortiert ist und die Schlüssel paarweise verschiedene nicht negative ganze Zahlen sind, so wachsen die Schlüssel mindestens so stark wie die Indices der Elemente.

Dann wird i in obiger while-Schleife maximal $\lfloor \log(k) \rfloor + 1$ mal verdoppelt.

Der Suchbereich $a[i/2], \ldots, a[i-1]$ hat somit maximal k Elemente.

Beispiel: Gesucht sei der Schlüssel $k = 24$ in der Folge

2	3	6	9	12	16	18	19	23	24	34	40	41	42	45	52	$60 \cdots$

Die while-Schleife terminiert nach der 4. Verdopplung von i, denn $a[15] = 52 > 24 = k$. Der Suchbereich $a[8], \cdots, a[15]$ kann nun mit binärer Suche oder der Fibonaccisuche nach dem Schlüssel k durchsucht werden.

Analyse: Wird der Bereich $a[i/2], \ldots, a[i-1]$ mit binärer Suche oder mit Fibonacci-Suche durchsucht, so findet man Schlüssel k stets in maximal $O(\log(k))$ Schritten.

Bemerkung: Das Verfahren ist geeignet, wenn k klein ist im Vergleich zu N.

Interpolationssuche

Idee: Schätze intuitiv die Position des Elementes mit Schlüssel k.

Kleine Schlüssel liegen am Anfang, große Schlüssel weiter hinten.

Inspiziere als nächstes immer das Element auf Position

$$m = l + \left\lfloor \frac{k - a[l].k}{a[r].k - a[l].k}(r - l + 0.999999...) \right\rfloor,$$

wobei l die linke Grenze und r der rechte Grenze des Suchbereichs ist.

Algorithmische Umsetzung:

Bei diesem Beispiel sei zu erwähnen, dass hier kein Wert auf die Berücksichtigung der Typkonvertierungen der jeweiligen Programmiersprache gelegt wird.

```
(1)   bool ip_Suche(Feld a[], int l, int r, int k)
(2)   {
(3)     if (l = r) {
(4)         if (a[l].k = k) {
(5)             return(true); }
(6)         else {
(7)             return(false); }}
(8)     int m := l + ⌊((k − a[l].k) / (a[r].k − a[l].k)) · (r − l + 0.999999...)⌋;
(9)     if ((m > l) and (k ≤ a[m − 1].k)) {
(10)        return(ip_Suche(a, l, m − 1, k)); }
(11)    if ((m < r) and (k ≥ a[m + 1].k)) {
(12)        return(ip_Suche(a, m + 1, r, k)); }
(13)    return(ip_Suche(a, m, m, k));
(14)  }
```

Beispiel: Gesucht sei der Schlüssel $k = 16$ in der Folge

2	3	6	9	12	16	18	19	23	24	34	40	42	49

Für m ergibt sich folgende Gleichung, wenn $l = 0$ und $r = (N - 1) = 13$ die Feldgrenzen sind:

$$m = 0 + \left\lfloor \frac{(16 - 2)}{(49 - 2)} (13 - 0 + 0.999999...) \right\rfloor,$$

Daraus ergibt sich $m = 4$. Da $a[4] = 12 < 16 = k$, suchen wir rekursiv weiter mit $l = 5$ und $r = 13$:

$$m = 5 + \left\lfloor \frac{(16-16)}{(49-16)} (13 - 5 + 0.999999) \right\rfloor,$$

Daraus ergibt sich $m = 5$. Da $a[5] = 16 = k$, endet die Suche erfolgreich.

Analyse: Man kann zeigen, dass Interpolationssuche im Mittel $\log(\log(N)) + 1$ Schlüsselvergleiche ausführt.

Bemerkung: Die Interpolationssuche benötigt in äquidistanten Schlüsselfolgen aufgrund der Linearität nur $\Theta(1)$ Schritte.

Beachte: Der Vorteil der geringen Anzahl von Schlüsselvergleichen geht leicht durch die auszuführenden arithmetischen Operationen verloren.

Selbstanordnende Listen

Das Ziel ist es, die Elemente, auf die häufig zugegriffen wird, möglichst weit vorne und die Elemente, auf die selten zugegriffen wird, am Ende der Liste zu platzieren.

Strategien: Sei k das Element, auf das zugegriffen wird.

1. Move-to-front-Regel: Mache das Element k zum ersten Element der Liste. Die Anordnung der übrigen Elemente bleibt erhalten.
2. Transpose-Regel: Vertausche k mit seinem Vorgänger.
3. Frequently-Count-Regel: Ordne die Elemente nach ihrer Zugriffshäufigkeit.

Experimentell ermittelte Messergebnisse für reale Daten ergaben:

- die T-Regel ist schlechter als die FC-Regel,
- die FC-Regel und MF-Regel sind etwa gleich gut,
- die MF-Regel ist allerdings in manchen Fällen besser.

Beispiel 1: Gegeben ist eine Liste mit folgenden Einträgen:

$$1, 2, 3, 4, 5, 6, 7$$

Betrachtet werden die folgenden Anfragen:

$$4, 2, 4, 5, 2, 2, 7, 4, 4$$

- Move-to-front-Regel

Anfrage	Folge							Vergleiche
	1	2	3	4	5	6	7	
4	4	1	2	3	5	6	7	4
2	2	4	1	3	5	6	7	3
4	4	2	1	3	5	6	7	2
5	5	4	2	1	3	6	7	5
2	2	5	4	1	3	6	7	3
2	2	5	4	1	3	6	7	1
7	7	2	5	4	1	3	6	7
4	4	7	2	5	1	3	6	4
4	4	7	2	5	1	3	6	1
								Summe 30

- Transpose-Regel

Anfrage	Folge							Vergleiche
	1	2	3	4	5	6	7	
4	1	2	4	3	5	6	7	4
2	2	1	4	3	5	6	7	2
4	2	4	1	3	5	6	7	3
5	2	4	1	5	3	6	7	5
2	2	4	1	5	3	6	7	1
2	2	4	1	5	3	6	7	1
7	2	4	1	5	3	7	6	7
4	4	2	1	5	3	7	6	2
4	4	2	1	5	3	7	6	1
								Summe 26

- Frequently-Count-Regel

Anfrage	Folge							Vergleiche
	$1^{[0]}$	$2^{[0]}$	$3^{[0]}$	$4^{[0]}$	$5^{[0]}$	$6^{[0]}$	$7^{[0]}$	
4	$4^{[1]}$	$1^{[0]}$	$2^{[0]}$	$3^{[0]}$	$5^{[0]}$	$6^{[0]}$	$7^{[0]}$	4
2	$4^{[1]}$	$2^{[1]}$	$1^{[0]}$	$3^{[0]}$	$5^{[0]}$	$6^{[0]}$	$7^{[0]}$	3
4	$4^{[2]}$	$2^{[1]}$	$1^{[0]}$	$3^{[0]}$	$5^{[0]}$	$6^{[0]}$	$7^{[0]}$	1
5	$4^{[2]}$	$2^{[1]}$	$5^{[1]}$	$1^{[0]}$	$3^{[0]}$	$6^{[0]}$	$7^{[0]}$	5
2	$4^{[2]}$	$2^{[2]}$	$5^{[1]}$	$1^{[0]}$	$3^{[0]}$	$6^{[0]}$	$7^{[0]}$	2
2	$2^{[3]}$	$4^{[2]}$	$5^{[1]}$	$1^{[0]}$	$3^{[0]}$	$6^{[0]}$	$7^{[0]}$	2
7	$2^{[3]}$	$4^{[2]}$	$5^{[1]}$	$7^{[1]}$	$1^{[0]}$	$3^{[0]}$	$6^{[0]}$	7
4	$2^{[3]}$	$4^{[3]}$	$5^{[1]}$	$7^{[1]}$	$1^{[0]}$	$3^{[0]}$	$6^{[0]}$	2
4	$4^{[4]}$	$2^{[3]}$	$5^{[1]}$	$7^{[1]}$	$1^{[0]}$	$3^{[0]}$	$6^{[0]}$	2
								Summe 28

  In den hochgestellten, eckigen Klammern speichern wir, wie häufig auf das Element bereits zugegriffen wurde.

Beispiel 2:
Gegeben ist eine Liste mit folgenden Einträgen:

$$1, 2, 3, 4, 5, 6, 7$$

Betrachte nun die folgenden Anfragen:

1. Greife 10 mal in der Reihenfolge $1, 2, 3, 4, 5, 6, 7$ zu.
   Anzahl der Vergleiche mit der MF-Regel = $1+2+3+4+5+6+7+9 \cdot 7 \cdot 7 = 469$.
   Das ergibt eine Anzahl von Vergleichen pro Zugriff von $\frac{469}{70} = 6.7$.
2. Greife zehn mal auf 1 zu, dann zehn mal auf 2 zu, usw.
   Anzahl der Vergleiche mit der MF-Regel = $10+11+12+13+14+15+16 = 91$.
   Das ergibt eine Anzahl von Vergleichen pro Zugriff von $\frac{91}{70} = 1.3$.

Mit der FC-Regel werden in beiden Fällen $(1 + 2 + 3 + 4 + 5 + 6 + 7) \cdot 10 = 10 + 20 + 30 + 40 + 50 + 60 + 70 = 280$ Vergleiche durchgeführt. Dabei ergeben sich Durchschnittskosten von $\frac{10+20+30+40+50+60+70}{70} = 4$.

Die FC-Regel hat den Nachteil, dass sie zusätzlichen Speicherplatz benötigt.

Es lässt sich zeigen, dass die MF-Regel höchstens doppelt so schlecht ist, wie jede andere Regel. (Amortisierte Laufzeitanalyse.)

3.4 Textsuche

Häufig wird in einer Folge von Zeichen (in einem Text) eine Zeichenkette (ein Muster) gesucht (Pattern Matching).

Man interessiert sich entweder für ein oder für alle Vorkommen des Musters im Text.

Gegeben:

1. Alphabet Σ
2. Text $T \in \Sigma^*$, $T = a_0, \ldots, a_{N-1}$ mit $a_i \in \Sigma$
3. Muster $P \in \Sigma^*$, $P = b_0, \ldots, b_{M-1}$ mit $b_i \in \Sigma$

Gesucht: $i \in \mathbb{N}_0$ mit

1. $0 \leq i \leq N - M$ und i ist die Anfangsposition des Musters P im Text T, d.h., $a_i = b_0, a_{i+1} = b_1, \ldots, a_{i+M-1} = b_{M-1}$,
2. $i = N$ falls das Muster P nicht im Text T enthalten ist.

Einfache Suche:

Methode: Durchsuche den gesamten Text schrittweise von links nach rechts und vergleiche bei jeder Position von links das Muster zeichenweise mit dem Text.

Beispiel:

```
Als Frauen verkleidete Gauner sind meist leicht zu erkennen.
erkennen
^
 erkennen
 ^
  erkennen
  ^
...
             erkennen
             ^
             erkennen
              ^
             erkennen
```

```
              ^
            erkennen
               ^
             erkennen
             ^
...
                                                        erkennen
                                                              ^
```

Analyse:

Im ungünstigsten Fall werden $(N - M + 1) \cdot M \in O(N \cdot M)$ viele Vergleiche durchgeführt.

Das Verfahren von Knuth-Morris-Pratt:

Methode: Vergleiche das Zeichen im Muster auf Position j mit dem Zeichen im Text auf Position i.
Wenn das Zeichen im Muster auf Position j mit dem Zeichen im Text auf Position i übereinstimmt, dann erhöhe i und j um eins.
Wenn das Zeichen im Muster auf Position j nicht mit dem Zeichen im Text auf Position i übereinstimmt, dann setze j auf die größte Position l zwischen 0 und $j - 1$ zurück, so das $b_0, \ldots, b_{l-1} = a_{i-l}, \ldots, a_{i-1}$.

Beispiel:

```
          i                 i                 i
T   ...011011...    ...011011...    ...011011...
P      011010           011010           011010
          j                 j               j
```

Dies wird wie folgt realisiert:

Sei `next[j]` die von rechts her nächste Stelle im Muster, die man mit der i-ten Position im Text vergleichen muss, falls $T[i]$ und $P[j]$ verschieden sind.

Beispiel:

Das Array `next[]` wird für das Muster in einer Vorphase vorausberechnet.

P	=	0	1	0	1	0	1	1	0	0
j	=	0	1	2	3	4	5	6	7	8
`next[j]`	=	-1	0	0	1	2	3	4	0	1

Berechnung des `next`-Arrays

```
(1)   void init_next_array (int next[], char P[], int M)
(2)   {
(3)     if (M > 0) {
(4)         int i := 0;
(5)         int j := −1;
(6)         next[0] := −1;
(7)         while (i < M − 1) {
(8)             if ((j = −1) or (P[i] = P[j])) {
(9)                 i + +;
(10)                j + +;
(14)                next[i] := j; }
(15)            else {
(16)                j := next[j]; } } }
(17)  }
```

Textsuche mit dem berechneten `next`-Array

```
(1)   int kmp_search (char T[], int N, char P[], int M, int next[])
(2)   {
(3)     if (M > 0) {
(4)         int i := 0;
(5)         int j := 0;
(6)         while ((i < N) and (j < M)) {
(7)             if ((j = −1) or (T[i] = P[j])) {
(8)                 i + +;
(9)                 j + +; }
(10)            else {
(11)                j := next[j]; } }
(12)        if (j = M) {
(13)            return(i − M); } }
(14)    return(N);
(15)  }
```

Analyse:

Variable j kann nur so oft erniedrigt werden, wie sie zuvor erhöht wurde.

Variable j wird immer zusammen mit Variable i erhöht.

Also wird j nur so oft erniedrigt, wie i erhöht wird.

Ist das `next`-Array bekannt, so hat das Verfahren eine lineare Laufzeit $O(N)$.

Da das `next`-Array in $O(M)$ Schritten aufgebaut werden kann, benötigt der gesamte Algorithmus $O(N + M)$ Schritte.

Verbessertes `next`-Array

```
(1)   void init_next_array (int next[], char P[], int M)
(2)   {
(3)     if (M > 0) {
(4)         int i := 0;
(5)         int j := −1;
(6)         next[0] := −1;
(7)         while (i < M − 1) {
(8)             if ((j = −1) or (P[i] = P[j])) {
(9)                 i + +;
(10)                j + +;
(11)                if (P[i] = P[j]) {
(12)                    next[i] := next[j]; }
(13)                else {
(14)                    next[i] := j; } }
(15)            else {
(16)                j := next[j]; } } }
(17)  }
```

P	=	0	1	0	1	0	1	1	0	0
j	=	0	1	2	3	4	5	6	7	8
`next'[j]`	=	-1	0	-1	0	-1	0	4	-1	1

3.5 Das Verfahren von Boyer-Moore:

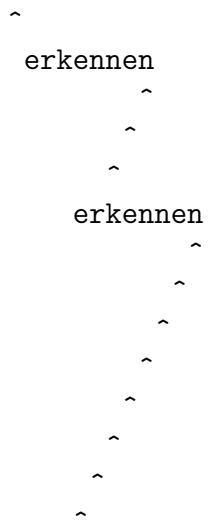
Methode: Bei diesem Verfahren werden die Zeichen im Muster von rechts nach links mit den Zeichen im Text verglichen.

Wenn es beim Vergleich der Zeichen im Text mit den Zeichen im Muster zu einem Mismatch kommt, dann verschiebe das Muster so weit wie möglich nach rechts, jedoch maximal um die Musterlänge.

```
(1)   int bm_suche (char T[], int N, char P[], int M)
(2)   {
(3)      int i := M − 1;
(4)      int j := M − 1;
(5)      while ((j ≥ 0) and (i ≤ N − 1)) {
(6)           if (T[i] = P[j]) {
(7)                i − −;
(8)                j − −; }
(9)           else {
(10)               i := i + „Musterverschiebung“; }
(11)               j := M − 1; } }
(12)     if (j < 0) {
(13)          return (i + 1); }
(14)     else
(15)          return (N); }
(16)  }
```

Beispiel:

```
Als Frauen verkleidete Gauner sind meist leicht zu erkennen.
erkennen
       ^
        erkennen
               ^
                erkennen
                       ^
                        erkennen
                               ^
                                erkennen
                                       ^
                                        erkennen
```



Die vereinfachte Heuristik nach Boyer und Moore:

Eine Verschiebung des Musters um 1 entspricht einer Erhöhung von i um $(M - 1 - j) + 1$ also um $M - j$.

Eine Verschiebung des Musters um k entspricht einer Erhöhung von i um $(M - 1 - j) + k$.

Der sogenannte delta-1-Wert (bad character shift) beschreibt für jedes Zeichen c im Text das erste Vorkommen des Zeiches im Muster (von rechts gezählt). Kommt c im Muster nicht vor, so ist delta-1$(c) = M$.

Beispiel mit Alphabet $\Sigma = \{a, b, c, d\}$:

$$P = a \; b \; c \; a \; b \; a$$

$$\begin{aligned} \text{delta-1}(a) &= 0 \\ \text{delta-1}(b) &= 1 \\ \text{delta-1}(c) &= 3 \\ \text{delta-1}(d) &= 6 \end{aligned}$$

In dem vereinfachten Verfahren von Boyer und Moore wird immer dann, wenn es zu einem Mismatch kommt, i auf $i + \max\{(M - 1 - j) + 1, \text{delta-1}(T[i])\}$ und j auf $j + (M - 1 - j) = M - 1$ gesetzt.

Bemerkungen:

Kommt es zu einem Mismatch zwischen $T[i]$ und $P[j]$, dann wird das Muster so weit nach rechts geschoben, bis das Zeichen $T[i]$ im Text über dem rechtesten Vorkommen von $T[i]$ im Muster steht.

Kommt das Zeichen $T[i]$ im Muster gar nicht vor, dann wird das Muster so weit nach rechts geschoben, bis das Zeichen $T[i]$ im Text vor dem ersten Zeichen im Muster steht.

Das Muster wird jedoch mindestens um eine Position nach rechts geschoben ($i := i + M - j$).

Obwohl obiges Verfahren eine worst-case-Laufzeit von $O(N \cdot M)$ hat ($T = 00 \cdots 0$, $P = 100 \cdots 0$), hat es sich in der Praxis ausgezeichnet bewährt.

Für genügend kurze Muster und hinreichend große Alphabete werden im Mittel $O(N/M)$ viele Schritte durchgeführt.

Das Muster wird dann also nahezu immer um seine Gesamtlänge nach rechts verschoben.

Kleine Modifikationen

1. Der delta-1-Wert für das letzte Zeichen $P[M-1]$ im Muster P kann auch auf das zweite Vorkommen des Zeichens im Muster (von rechts gezählt) gesetzt werden.

   Beispiel mit Alphabet $\Sigma = \{a, b, c, d\}$:

   $$P = a \;\; b \;\; c \;\; a \;\; b \;\; a$$

   $$\begin{aligned} \text{delta'-1}(a) &= 2 \\ \text{delta'-1}(b) &= 1 \\ \text{delta'-1}(c) &= 3 \\ \text{delta'-1}(d) &= 6 \end{aligned}$$

2. Bei einem Mismatch kann i in Zeile (10) auch um

   $(M - 1 - j) + \text{delta'-1}(P[M-1])$, falls $j < M - 1$, bzw. um
   $(M - 1 - j) + \text{delta-1}(T[i + M - j])$ erhöht werden.

   Im ersten Fall wird das Muster so weit nach rechts geschoben, dass das zweite Vorkommen vor rechts des letzten Zeichens im Muster unter der Position $i + (M - 1 - j)$ im Text steht.

   Im zweiten Fall wird das Zeichens im Text auf Position $i + (M - 1 - j) + 1$ betrachtet und dieses unter sein erstes Vorkommen vor rechts im Muster gesetzt.

Verbesserung des einfachen Verfahrens von Boyer und Moore

Die Sub-Muster im Muster können ähnlich wie bei dem Verfahren von Knuth Morris und Pratt mit berücksichtigt werden.

Der sogenannte delta-2-Wert beschreibt, um welchen Wert i maximal erhöht werden kann, wenn an einer Position j im Muster ein Mismatch eintritt.

Genutzt wird die Information, dass auf den Positionen $i+1, \ldots, i+(M-1-j)$ im Text die gleichen Zeichen stehen, wie im Muster auf den Positionen $j, \ldots, M-1$. Wird zusätzlich noch die Information genutzt, dass auf Position i im Text ein anderes Zeichen steht als auf Position j im Muster, dann erhalten wir den verbesserten delta-2-Wert.

Beispiel für den einfachen delta-2-Wert.

P	$=$	1	0	0	1	0	1	0	1	0
j	$=$	0	1	2	3	4	5	6	7	8
delta-2(j)	$=$	15	14	13	7	6	5	4	3	1

```
                    i
T ..................x..........
P           100101010
                    j           j = 8

                     i          i := i + 1
T ..................x..........
P            100101010
                     j          j := M-1
```

```
                   i
T .................x0..........
P           100101010
                   j            j = 7

                      i         i := i + 3
T .................x0..........
P             100101010
                      j         j := M-1
```

```
                  i
T ................x10..........
P           100101010
```

```
                  j             j = 6

                      i         i := i + 4
T ................x10..........
P             100101010
                      j         j := M-1
```

```
                 i
T ...............x010..........
P           100101010
                 j              j = 5

                      i         i := i + 5
T ...............x010..........
P             100101010
                      j         j := M-1
```

```
                i
T ..............x1010..........
P           100101010
                j               j = 4

                      i         i := i + 6
T ..............x1010..........
P             100101010
                      j         j := M-1
```

```
               i
T .............x01010..........
P           100101010
               j                j = 3

                      i         i := i + 7
T .............x01010..........
P             100101010
                      j         j := M-1
```

```
              i
T ............x101010..........
P           100101010
              j                 j = 2

                           i    i:= i + 13
T ............x101010..........
P                 100101010
                           j    j := M-1
```

```
             i
T ...........x0101010.........
P           100101010
             j                  j = 1

                           i    i:= i + 14
T ...........x0101010.........
P                  100101010
                           j    j := M-1
```

```
            i
T ..........x00101010.........
P           100101010
            j                  j = 0

                           i    i:= i + 15
T ..........x00101010.........
P                  100101010
                           j    j := M-1
```

Beispiel für den verbesserten delta-2-Wert.

P	$=$	1	0	0	1	0	1	0	1	0
j	$=$	0	1	2	3	4	5	6	7	8
delta-2(j)	$=$	15	14	13	7	11	7	9	7	1

```
                   i
T ..................x.......... x nicht 0
P           100101010
                   j           j = 8

                    i          i := i + 1
T ...................x.........
P            100101010
                    j          j := M-1
```

```
                   i
T .................x0.......... x nicht 1
P           100101010
                   j           j = 7

                          i    i := i + 7
T .................x0.........
P                 100101010
                          j    j := M-1
```

```
                  i
T ................x10.......... x nicht 0
P           100101010
                  j             j = 6

                           i    i := i + 9
T ................x10..........
P                  100101010
                           j    j := M-1
```

```
                 i
T ................x010.......... x nicht 1
P           100101010
                 j              j = 5

                        i       i := i + 7
T ................x010..........
P               100101010
                        j       j := M-1
```

```
                i
T ...............x1010.......... x nicht 0
P           100101010
                j               j = 4

                           i    i := i + 11
T ...............x1010..........
P                  100101010
                           j    j := M-1
```

```
               i
T ..............x01010.......... x nicht 1
P           100101010
                j               j = 3

                      i         i := i + 7
T ..............x01010..........
P              100101010
                      j         j := M-1
```

```
              i
T .............x101010.......... x nicht 0
P           100101010
              j                 j = 2

                           i    i:= i + 13
T .............x101010..........
P                  100101010
                           j    j := M-1
```

```
             i
T ...........x0101010.......... x nicht 0
P            100101010
             j                  j = 1

                                i    i:= i + 14
T ...........x0101010..........
P                    100101010
                                j    j := M-1
```

```
            i
T ..........x00101010.......... x = nicht 1
P           100101010
            j                   j = 0

                                i    i:= i + 15
T ..........x00101010..........
P                    100101010
                                j    j := M-1
```

In dem erweiterten Verfahren von Boyer und Moore wird immer dann, wenn es zu einem Mismatch auf Position j im Muster kommt, i auf $i + \max\{\text{delta'-1}(T[i]), \text{delta-2}[j]\}$ und j auf $M-1$ gesetzt.

Bemerkungen:

Da der delta-2-Wert mindestens $M - j$ ist, kann $M - j$ (Musterverschiebung um 1) aus der Maximumbildung herausgenommen werden.

```
(1)   int bm_suche (char T[], int N, char P[], int M)
(2)   {
(3)      int i := M − 1;
(4)      int j := M − 1;
(5)      while ((j ≥ 0) and (i ≤ N − 1)) {
(6)           if (T[i] = P[j]) {
(7)                i − −;
(8)                j − −; }
(9)           else {
(10)               i := i + max{delta'-1(T[i]), delta-2(j)};
(11)               j := M − 1; } }
(12)     if (j < 0) {
(13)          return (i + 1); }
```

```
(14)     else
(15)          return (N); }
(16)  }
```

Diese Verbesserung beseitigt zwar die worst-case Instanzen bei der Textsuche (hier ohne Beweis), bringt aber in der Praxis eigentlich nichts. Deshalb wird in der Regel mit der vereinfachten Version gearbeitet.

3.6 Hashverfahren

Beim Hashing werden die Schlüssel zur Berechnung einer Position in einem Feld benutzt.

Sei K eine Schlüsselmenge und I eine endliche Indexmenge
(in der Regel $I = \{0, 1, 2 \ldots, M-1\}$).

Die Hashfunktion

$$h : K \to I$$

wird zum Suchen und zum Platzieren der Elemente eingesetzt.

Sie soll die Schlüssel gleichmäßig auf die Indizes verteilen und den Indexbereich ausschöpfen.

Sie soll schnell zu berechnen sein.

Wenn mehrere Schlüssel auf denselben Index abgebildet werden, spricht man von einer Kollision.

Bei einer Kollision muss ein Ersatzplatz gefunden werden.

Welche Funktionen sind als Hashfunktion geeignet?

In der Praxis hat sich Ausblenden und Restbildung bewährt.

Ausblenden: Benutze nur einen Teil des Schlüssels (nicht alle Stellen).

Restbildung: Der verbleibende Schlüssel k wird ganzzahlig durch die Länge der Hashtabelle dividiert. Der Rest wird als Index verwendet.

$$h(k) = k \bmod |I|$$

Die besten Ergebnisse erhält man, wenn $|I|$ eine Primzahl ist.

Schlüssel, die nicht als Zahlen interpretiert werden können, müssen vorher geeignet umgerechnet werden.

Beispiel:

Schlüssel sind Zeichenketten der Länge 2.

Zeichen werden als Zahlen interpretiert.

Die entstandenen Zahlen werden zum Schlüssel addiert.

Zeichenkette	k_1	k_2	$k = k_1 + k_2$	$i = k \bmod 7$
GL	71	76	147	0
LX	76	88	164	3
SL	83	76	159	5
GT	71	84	155	1
RX	82	88	170	2
GX	71	88	159	5

Universelle Hashfunktionen

Gesucht sind Hashfunktionen, die im Mittel akzeptabel sind.

Sei $M = |I|$ und H eine endliche Menge von $\geq M$ Hashfunktionen. H heißt universell, wenn für jedes Paar von verschiedenen Schlüsseln $x, y \in K$ gilt:

$$|\{h \in H \mid h(x) = h(y)\}| \leq \left\lceil \frac{|H|}{M} \right\rceil$$

Für jedes Schlüsselpaar x, y führt höchstens der M-te Teil der Hashfunktionen die beiden Schlüssel x und y zu einer Kollision.

Die Wahrscheinlichkeit, dass zwei Schlüssel mit einer Hashfunktion aus H auf den gleichen Index abgebildet werden ist $\frac{1}{M}$.

Sei $p \geq M$ eine Primzahl und

$$h_{a,b} : \{0, \ldots, p-1\} \to \{0, \ldots, M-1\}$$

mit

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod M.$$

Dann ist

$$\{h_{a,b} \mid 1 \leq a < p,\ 0 \leq b < p\}$$

eine universelle Klasse von Hashfunktionen (lineare Algebra).

Kollisionsauflösung:

Probleme treten auf:

- bei der Platzierung, wenn der Eintrag auf der berechneten Hashadresse nicht leer ist,
- bei der Suche, wenn der berechnete Platz ein anderes Element enthält.

Zur Kollisionsauflösung muss ein Ersatzplatz gefunden werden.

Verkettung der Überläufer:

Die Überläufer können zum Beispiel in einer linearen Liste verkettet werden, welche an den Hashtabelleneintrag angehängt wird, der sich aus der Hashfunktion angewendet auf den Schlüssel ergibt.

Bei einer erfolglosen Suche nach Schlüssel k betrachten wir alle Elemente in der Liste an $h(k)$.

Die Durchschnittliche Anzahl der Einträge in $h(k)$ ist N/M, wenn N Einträge auf M Listen verteilt sind.

Belegungsfaktor:

$$\alpha = \frac{N}{M}$$

Im Mittel ist die Anzahl der bei der erfolglosen Suche nach einem Schlüssel betrachteten Einträge also

$$C'_n = \frac{N}{M} = \alpha$$

Beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge gerade $(j-1)/M$.

Also betrachten wir bei einer späteren Suche nach dem j-ten Schlüssel gerade $1 + (j-1)/M$ Einträge im Durchschnitt, wenn stets am Listenende eingefügt und kein Datensatz entfernt wurde.

Im Mittel ist die Anzahl der bei der erfolgreichen Suche nach einem Schlüssel betrachteten Einträge also

$$C_n = \frac{1}{N}\sum_{j=1}^{N}(1 + (j-1)/M) = 1 + \frac{N-1}{2M} \approx 1 + \frac{\alpha}{2}$$

Offene Hashverfahren:

Speichere die Überläufer in der Hashtabelle und nicht in zusätzlichen Listen. Nur sinnvoll für $\alpha < 1$.

Ist die Hashadresse $h(k)$ belegt, so wird eine Ausweichposition gesucht.

Die Folge der zu betrachtenden Speicherplätze für einen Schlüssel nennt man Sondierungsfolge.

Sei $h : K \to \{0, \ldots, M-1\}$ eine Hashfunktion und $s : \{0, \ldots, M-1\} \times K \to \mathbb{N}_0$ eine Funktion, so dass für jedes $k \in K$, die Folge $(h(k) - s(j,k)) \bmod M$ für $j = 0, 1, \ldots, M-1$ eine Permutation aller Hashadressen $0, \ldots, M-1$ ist, dann ist s eine Sondierungsfunktion für h.

Lineares Sondieren:

Beim linearen Sondieren ist für Schlüssel k die Sondierungsfolge

$$h(k), h(k)-1, h(k)-2, \ldots, 0, M-1, \ldots, h(k)+1$$

Die Sondierungsfunktion ist somit

$$s(j,k) = j$$

Beispiel:

$M = 7$,
$K = \{0, \ldots, 500\}$,
$h(k) = k \bmod 7$,
$s(j,k) = j$ (lineares Sondieren)

Einfügen der Schlüssel 12, 53, 5, 15, 2, 19.

0	1	2	3	4	5	6
					12	
				53	12	
			5	53	12	
	15		5	53	12	
	15	2	5	53	12	
19	15	2	5	53	12	

Nachteil:

Das Verfahren neigt zur primären Häufung, indem gewisse Bereiche keine Lücke mehr aufweisen.

Quadratisches Sondieren:

Es wird um $h(k)$ herum mit quadratisch wachsendem Abstand nach einem freien Platz gesucht.

Sondierungsfolge:

$h(k), (h(k)+1) \bmod M, (h(k)-1) \bmod M, (h(k)+4) \bmod M, (h(k)-4) \bmod M, \ldots$

Sondierungsfunktion:

$$s(j,k) = (\lceil j/2 \rceil)^2 (-1)^j$$

Wenn M ein Primzahl der Form $4i+3$, $i \in \mathbb{N}$, ist, dann ist garantiert, dass die Sondierungsfolge Modulo M eine Permutation der Hashadressen 0 bis $M-1$ ist (ohne Beweis).

Gegenbeispiel: Für $M = 13$ ist die Sondierungsfolge keine Permutation der Hashadressen 0 bis $M-1$.

$$
\begin{aligned}
k \bmod 13 - (\lceil 0/2 \rceil)^2(-1)^0 \bmod 13 &= k \bmod 13 \\
k \bmod 13 - (\lceil 1/2 \rceil)^2(-1)^1 \bmod 13 &= k+1 \bmod 13 \\
k \bmod 13 - (\lceil 2/2 \rceil)^2(-1)^2 \bmod 13 &= k+12 \bmod 13 \\
k \bmod 13 - (\lceil 3/2 \rceil)^2(-1)^3 \bmod 13 &= k+4 \bmod 13 \\
k \bmod 13 - (\lceil 4/2 \rceil)^2(-1)^4 \bmod 13 &= k+9 \bmod 13 \\
k \bmod 13 - (\lceil 5/2 \rceil)^2(-1)^5 \bmod 13 &= k+9 \bmod 13 \\
k \bmod 13 - (\lceil 6/2 \rceil)^2(-1)^6 \bmod 13 &= k+4 \bmod 13 \\
k \bmod 13 - (\lceil 7/2 \rceil)^2(-1)^7 \bmod 13 &= k+3 \bmod 13 \\
k \bmod 13 - (\lceil 8/2 \rceil)^2(-1)^8 \bmod 13 &= k+10 \bmod 13 \\
k \bmod 13 - (\lceil 9/2 \rceil)^2(-1)^9 \bmod 13 &= k+12 \bmod 13 \\
k \bmod 13 - (\lceil 10/2 \rceil)^2(-1)^{10} \bmod 13 &= k+1 \bmod 13 \\
k \bmod 13 - (\lceil 11/2 \rceil)^2(-1)^{11} \bmod 13 &= k+10 \bmod 13 \\
k \bmod 13 - (\lceil 12/2 \rceil)^2(-1)^{12} \bmod 13 &= k+3 \bmod 13
\end{aligned}
$$

Beispiel:

$h(k) = k \bmod 7$,
$s(j,k) = (\lceil j/2 \rceil)^2(-1)^j$ (quadratisches Sondieren)

Einfügen der Schlüssel 12, 53, 5, 15, 2, 19.

0	1	2	3	4	5	6
					12	
				53	12	
				53	12	5
	15			53	12	5
	15	2		53	12	5
19	15	2		53	12	5

Nachteil:

Zwei Schlüssel k und k' mit $h(k) = h(k')$ durchlaufen stets dieselbe Sondierungsfolge. Sie behindern sich also auf Ausweichplätzen (sekundäre Häufung).

Eine Analyse der Effizienz des linearen und quadratischen Sondierens zeigt, dass für die durchschnittliche Anzahl der bei erfolgloser bzw. erfolgreicher Suche betrachteten Einträge C'_n bzw. C_n gilt:

1. Lineares Sondieren:

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \qquad C_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

2. Quadratisches Sondieren:

$$C'_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \qquad C_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

Vergleich der Effizienz zwischen linearem und quadratischem Sondieren:

	linear		quadratisch	
α	erfolgreich	erfolglos	erfolgreich	erfolglos
0.5	1.5	2.5	1.44	2.19
0.9	5.5	50.5	2.85	11.40
0.95	10.5	200.5	3.52	22.05
1	-	-	-	-

Verbesserte Kollisionsauflösung:

Schlüsselabhängige Sondierung (double Hashing):

Verwende für die Sondierungsfolge eine zweite Hashfunktion h'.

Sondierungsfolge:

$$h(k), (h(k) - h'(k)) \bmod M, (h(k) - 2 \cdot h'(k)) \bmod M, \ldots, (h(k) - (M-1) \cdot h'(k)) \bmod M$$

Sondierungsfunktion:

$$s(j, k) = j \cdot h'(k)$$

Die Funktion h' muss so gewählt werden, dass die Sondierungsfolge eine Permutation der Hashadressen ist. Das bedeutet, dass $h'(k) \neq 0$ sein muss und M nicht teilen darf.

Ist M eine Primzahl und $h(k) = k \bmod M$, so erfüllt $h'(k) = 1 + (k \bmod (M-2))$ die erwarteten Anforderungen.

Unterschiedliche Schlüssel, die auf denselben Index abgebildet werden, erhalten unterschiedliche Inkremente.

Beispiel:

$M = 7$, $h(k) = k \bmod 7$, $h'(k) = 1 + (k \bmod 5)$

Einfügen der Schlüssel 12, 53, 5, 15, 2, 19.

0	1	2	3	4	5	6
				53	12	
			5	53	12	
	15		5	53	12	
	15	2	5	53	12	
19	15	2	5	53	12	

Verbesserung der erfolgreichen Suche:

Die durchschnittliche Suchzeit bei der erfolgreichen Suche variiert bei Hashverfahren ohne Häufung mit unterschiedlicher Reihenfolge des Einfügens der Schlüssel.

In Fällen, in denen wesentlich häufiger gesucht wird als eingefügt, kann es lohnend sein, die Schlüssel beim Einfügen eines neuen Schlüssels so zu reorganisieren, dass die Suchzeit verkürzt wird.

Brents Algorithmus:

0	1	2	3	4	5	6
				53	12	
			5	53	12	
		12		53	5	

Wird Schlüssel 5 nach Inspektion der Plätze $5, 4, 3$ bei Hashadresse 3 eingetragen, so ist die durchschnittliche Suchzeit der drei eingetragenen Elemente $(1+1+3)/3 = 5/3 = 1.66$.

Man hätte aber auch Schlüssel 5 auf Platz 5 setzen können und Schlüssel 12 weiter sondieren lassen können.

Dann wäre die durchschnittliche Suchzeit $(1 + 1 + 2)/3 = 4/3 = 1.33$.

Methode: Einfügen eines Schlüssels k: Beginne mit Hashadresse $i = h(k)$. Falls Position i belegt ist, betrachte die beiden Hashadressen $b = (i - h'(k)) \bmod M$ und $b' = (i - h'(k')) \bmod M$, wobei k' der Schlüssel auf Hashadresse i ist. Ist b eine freie Hashadresse, dann positioniere Schlüssel k auf Position b, ansonsten, ist b' eine freie Hashadresse, dann positioniere Schlüssel k auf Position i und k' auf Position b', ansonsten fahre fort mit dem Versuch Schlüssel k auf Position $i = b$ zu positionieren.

0	1	2	3	4	5	6
				53	12	
		12		53	5	
	15	12		53	5	
	15	12		53	5	2
19	15	12		53	5	2

4 Suchbäume

4.1 Definitionen

Bäume sind verallgemeinerte Listenstrukturen. Ein Element (Knoten) hat nicht, wie bei einer linearen Liste nur einen Nachfolger (Sohn), sondern eine endliche, begrenzte Anzahl von Nachfolgern (Söhnen).

Der Knoten ohne Vorgänger (Vater) heißt Wurzel.

Die Knoten ohne Nachfolger heißen Blätter.

Knoten mit Nachfolger heißen innere Knoten.

Eine Folge von Knoten $u_1, \ldots, u_k$, wobei u_i mit $1 \leq i \leq k$ ein Nachfolger von u_{i-1} ist, heißt Weg der Länge k.

Bäume, in denen jeder Knoten genau k Nachfolger hat, heißen k-näre Bäume.

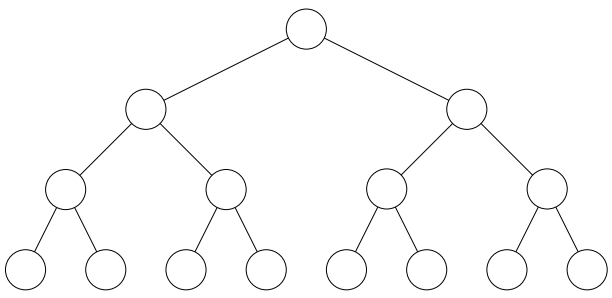
Die Höhe eines Baumes ist die Länge eines längsten Weges von der Wurzel zu einem Blatt -1. Dies ist die Anzahl der Nachfolger auf einem längsten Weg von der Wurzel zu einem Blatt. Ein Baum mit genau einem Knoten hat die Höhe 0.

Die Tiefe eines Knotens u ist die Länge des Weges von der Wurzel zu Knoten u -1. Dies ist die Anzahl der Nachfolger auf dem Weg von der Wurzel zu Knoten u. Die Wurzel hat die Tiefe 0.

Auf den Nachfolgern besteht in der Regel eine Ordnung, linker bzw. rechter Nachfolger, oder i-ter Nachfolger usw. In diesen Fällen sprechen wir von geordneten Bäumen.

Ein Baum heißt vollständig, wenn alle innere Knoten die maximal mögliche Anzahl von Nachfolgern und alle Blätter dieselbe Tiefe haben.

Ein vollständiger Binärbaum:



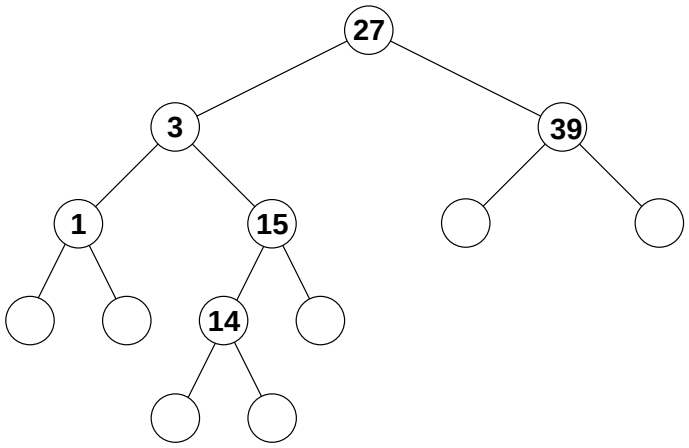
Suchbäume sind geordnete binäre Bäume, die in den inneren Knoten Schlüssel (oder Elemente mit Schlüsseln) speichern.

Sei u ein innerer Knoten, der den Schlüssel $k(u)$ speichert.

Dann sind alle Schlüssel im linken Teilbaum von u kleiner als $k(u)$ und alle Schlüssel im rechten Teilbaum von u größer als $k(u)$.

Die Blätter repräsentieren Intervalle zwischen den in den inneren Knoten gespeicherten Schlüsseln.

Beispiel:



Die Blätter speichern die Intervalle

$$(-\infty, 1), (1, 3), (3, 14), (14, 15), (15, 27), (27, 39), (39, \infty).$$

Ein Schlüssel x kann in einem Suchbaum wie folgt gefunden werden:

Beginne bei der Wurzel p und vergleiche x mit dem bei p gespeicherten Schlüssel $k(p)$; ist x kleiner als $k(p)$, setze die Suche mit dem linken Nachfolger von p fort. Ist x größer als $k(p)$, setze die Suche mit dem rechten Nachfolger von p fort.

Wird ein Blatt erreicht, ist Schlüssel x nicht im Suchbaum gespeichert.

Einfügen eines Schlüssels x in einem Suchbaum:

Suche den Schlüssel x im Suchbaum. Falls x nicht im Suchbaum ist, endet die Suche in einem Blatt. Füge dort den Schlüssel ein und erzeuge zwei neue Blätter.

Die Form des dabei entstehenden Suchbaumes hängt stark von der Einfügereihenfolge ab.

Entfernen eines Schlüssels x in einem Suchbaum:

Suche den Schlüssel x im Suchbaum. Falls x im Suchbaum ist, sei u der Knoten mit $k(u) = x$.

Sind beide Nachfolger von u Blätter, dann mache u zu einem Blatt.

Ist ein Nachfolger von u ein Blatt und der andere Nachfolger ein innerer Knoten, dann hänge den anderen Nachfolger von u an den Vater von u.

Sind beide Nachfolger von u innere Knoten, dann suche im rechten Teilbaum von u den Knoten v mit kleinstem Schlüssel $k(v)$. Schlüssel $k(v)$ ist immer größer als $k(u)$. Der Knoten v heißt symmetrischer Nachfolger von u. Ersetze $k(u)$ durch $k(v)$ und entferne Knoten v. Knoten u kann natürlich auch durch den symmetrischer Vorgänger von u ersetzt werden. Dies ist der Knoten v im linken Teilbaum von u mit größtem Schlüssel $k(v)$.

Durchlaufordnungen:

Jeder Durchlauf startet an der Wurzel:

Hauptreihenfolge: Betrachte zuerst den Knoten selbst, dann den linken Teilbaum, dann den rechten Teilbaum.

Nebenreihenfolge: Betrachte zuerst den linken Teilbaum, dann den rechten Teilbaum, dann den Knoten selbst.

Symmetrische Reihenfolge: betrachte zuerst den linken Teilbaum, dann den Knoten selbst, dann den rechten Teilbaum.

In unserem Beispiel:

Hauptreihenfolge: 27, 3, 1, 15, 14 ,39

Nebenreihenfolge: 1, 14, 15, 3, 39, 27

Symmetrische Reihenfolge: 1, 3, 14, 15, 27, 39

Ein Binärbaum mit N inneren Knoten hat $N + 1$ Blätter.

Seine Höhe ist maximal N und mindestens $\lceil \log(N + 1) \rceil$.

4.2 AVL-Bäume

Der Aufwand der Operationen Suchen, Einfügen und Entfernen ist im wesentlichen von der Höhe des Baumes abhängig.

Ziel: Die Höhe des Suchbaumes soll auf $O(\log(N))$ beschränkt werden, damit die Such- und Einfügeoperationen höchstens $O(\log(N))$ Vergleiche benötigen.

Ein Suchbaum ist höhenbalanciert, wenn für jeden inneren Knoten u sich die Höhe des rechten Teilbaumes von u von der Höhe des linken Teilbaumes von u höchstens um 1 unterscheidet.

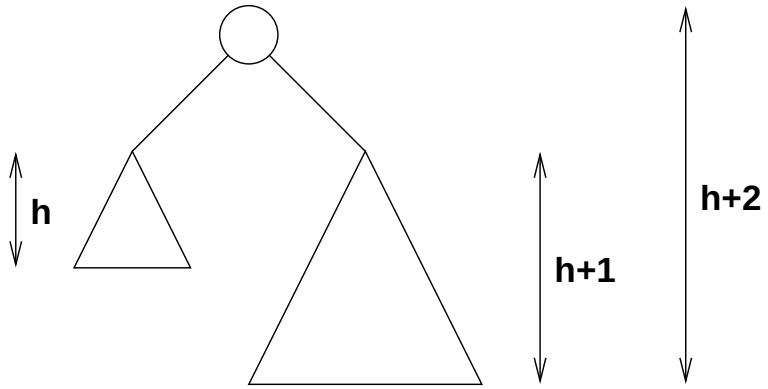
Höhenbalancierte Bäume werden auch AVL-Bäume genannt (nach den Erfindern Adelson-Velskij und Landis).

Die Höhenbedingung sichert, dass AVL-Bäume mit N inneren Knoten und $N+1$ Blättern eine Höhe aus $O(\log(N))$ haben.

Ein AVL-Baum der Höhe 1 mit minimaler Blattanzahl hat 2 Blätter.

Ein AVL-Baum der Höhe 2 mit minimaler Blattanzahl hat 3 Blätter.

Einen AVL-Baum der Höhe $h+2$ mit minimaler Blattanzahl erhält man, wenn einen AVL-Baum mit Höhe $h+1$ und einen AVL-Baum mit Höhe h, beide mit minimaler Blattanzahl, wie folgt zu einem Baum der Höhe $h+2$ zusammenfügt wird:



Ist F_i die i-te Fibonacci-Zahl, so hat ein AVL-Baum der Höhe h mindestens F_{h+2} Blätter.

Die Fibonacci-Zahlen wachsen exponentiell,

$$F_n \approx c \cdot 1.618^n$$

mit einer Konstanten c.

Die Anzahl der Blätter in einem höhenbalancierten Baum wächst also exponentiell mit der Höhe. Daraus folgt, dass ein AVL-Baum mit N Blättern (und $N-1$ inneren Knoten) eine Höhe aus $O(\log(N))$ hat. Denn jeder AVL-Baum mit Höhe h hat $N \geq F_{h+2} \approx c \cdot 1.618^{h+2}$ Blätter.

Anmerkung: Ist an jedem inneren Knoten u der Höhenunterschied zwischen den beiden Teilbäumen an u höchstens k, so ist die Anzahl der Blätten in einem Baum der Höhe $h+k$ mindestens doppelt so groß wie die Anzahl der Blätter in einem Baum der Höhe h. Somit ist $N - 1 \geq 2^{\lfloor \frac{h}{k} \rfloor}$ und $h \in O(\log(N))$.

Suchen:

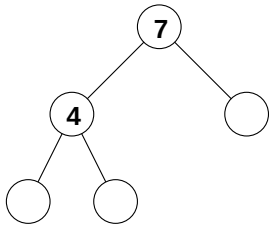
Das Suchen in AVL-Bäumen erfolgt wie in einem Suchbaum, und ist durch die AVL-Ausgeglichenheit in $O(\log(N))$ vielen Schritten möglich.

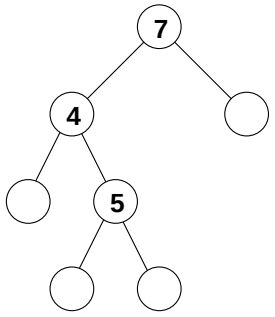
Einfügen:

Der Schlüssel x wird wie beim Einfügen in einem Suchbaum eingefügt.

Liegt anschließend kein AVL-Baum mehr vor, wird die AVL-Ausgeglichenheit wiederhergestellt.

Beispiel: Füge Schlüssel 5 ein.





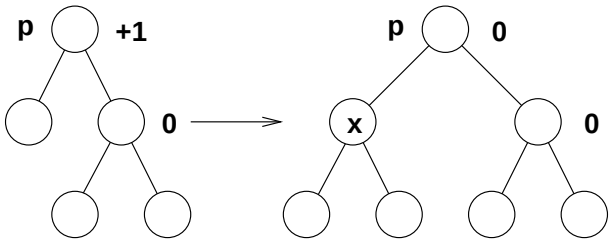
Um die Höhenbedingung an einen inneren Knoten u zu überprüfen, genügt es einen sogenannten Balancefaktor $bal(u)$ mitzuführen, der wie folgt definiert ist:

$bal(u)$ ist die Höhe des rechten Teilbaumes minus der Höhe des linken Teilbaumes.

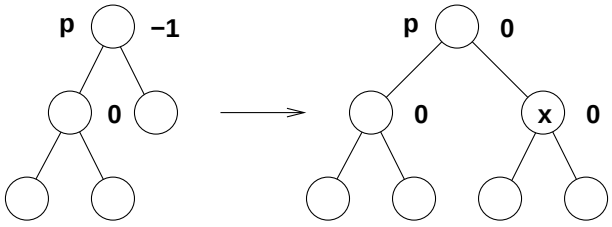
In einem AVL-Baum gilt für jeden inneren Knoten $bal(u) \in \{-1, 0, 1\}$.

Sei p der Vater des Blattes, bei dem die Suche endet.

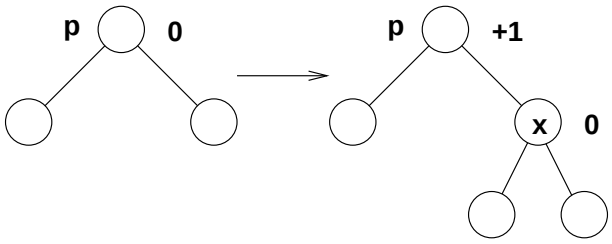
Fall 1: $bal(p) = +1$ und x ist kleiner als $k(p)$



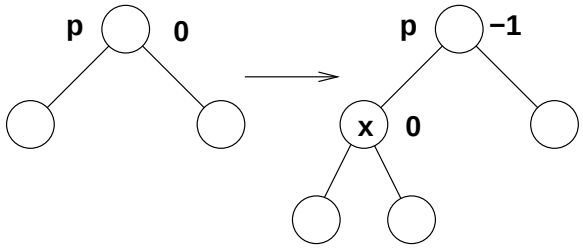
Fall 2: $bal(p) = -1$ und x ist größer als $k(p)$



Fall 3.1: $bal(p) = 0$ und x ist größer als $k(p)$



Fall 3.2: $bal(p) = 0$ und x ist kleiner als $k(p)$



Durch Einfügen eines neuen Knotens als rechten oder linken Sohn von p wird p ein Knoten mit Balancefaktor -1 oder $+1$. Dadurch hat sich die Höhe des Teilbaumes verändert.

Wir rufen eine Funktion `upin(p)` für den Knoten p auf, die den Suchpfad zurückläuft, die Balancefaktoren prüft, gegebenenfalls adjustiert und Umstrukturierungen (sogenannte Rotationen oder Doppelrotationen) vornimmt.

Invariante:

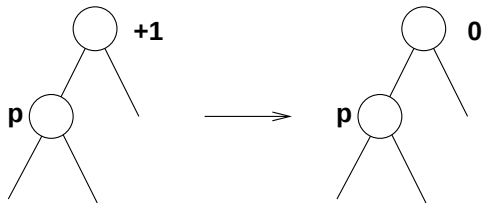
Wenn `upin(p)` aufgerufen wird, ist $bal(p) \in \{-1, +1\}$ und die Höhe des Teilbaumes mit Wurzel p um eins gewachsen.

`upin(p)` bricht ab, wenn p die Wurzel ist.

Wir unterscheiden zwei Fälle, je nachdem ob p linker oder rechter Sohn seines Vaters ist.

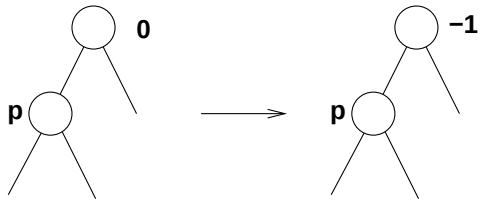
Fall 1: (p ist linker Sohn seines Vaters $\varphi(p)$)

Fall 1.1: ($bal(\varphi(p)) = +1$), x wird im linken oder rechten Teilbaum von p angehangen



Die Höhe des Teilbaumes hat sich nicht verändert, keine weitere Umstrukturierung notwendig.

Fall 1.2: ($bal(\varphi(p)) = 0$) x wird im linken oder rechten Teilbaum von p angehangen



Die Höhe des Teilbaumes hat sich verändert, weitere Umstrukturierung notwendig.

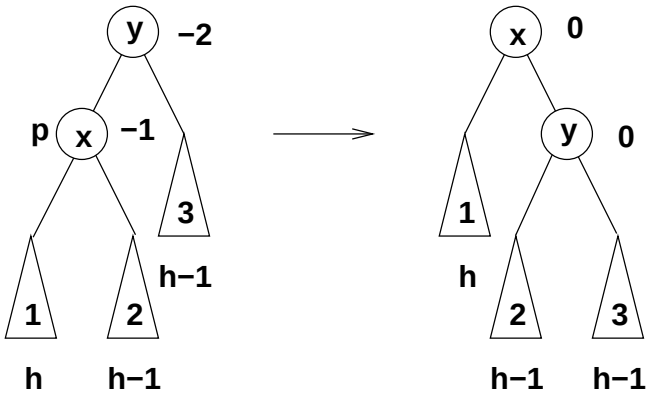
Fall 1.3: ($bal(\varphi(p)) = -1$)

Da $bal(\varphi(p)) = -1$, hatte der linke Teilbaum von $\varphi(p)$ bereits vor dem Einfügen des neuen Schlüssels eine um 1 größere Höhe als der rechte Teilbaum von $\varphi(p)$.

Da der linke Teilbaum in der Höhe um eins gewachsen ist, ist die AVL-Ausgeglichenheit bei $\varphi(p)$ verletzt.

Wir strukturieren um.

Fall 1.3.1: ($bal(p) = -1$)

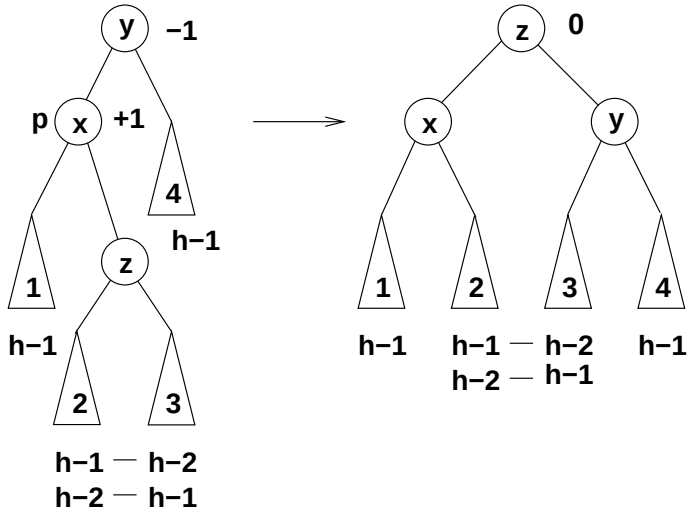


Nach Voraussetzung ist der Teilbaum mit Wurzel p um eins gewachsen und der linke Teilbaum von $\varphi(p)$ um eins größer als der rechte Teilbaum von $\varphi(p)$.

Eine Rotation nach rechts bringt den Baum bei $\varphi(p)$ wieder in die Balance. Dabei wird der rechte Teilbaum von p neuer linker Teilbaum von $\varphi(p)$ und $\varphi(p)$ mit rechtem Teilbaum wird neuer rechter Teilbaum von p.

Da nach der Rotation der Teilbaum mit Wurzel $\varphi(p)$ nicht um eins gewachsen ist, ist eine weitere Umstrukturierung nicht notwendig.

Fall 1.3.2: $(bal(p) = +1)$



Entweder sind die Teilbäume 2 und 3 beide leer oder die einzig möglichen Höhenkombinationen für die Teilbäume 2 und 3 sind $(h-1, h-2)$ und $(h-2, h-1)$.

Falls nicht beide Teilbäume leer sind, können sie nicht die gleiche Höhe haben. Denn aufgrund der Invariante ist der Teilbaum mit Wurzel p in der Höhe um eins gewachsen und wegen der Annahme von Fall 1.3.2 ist der rechte Teilbaum von p um 1 höher als sein linker Teilbaum.

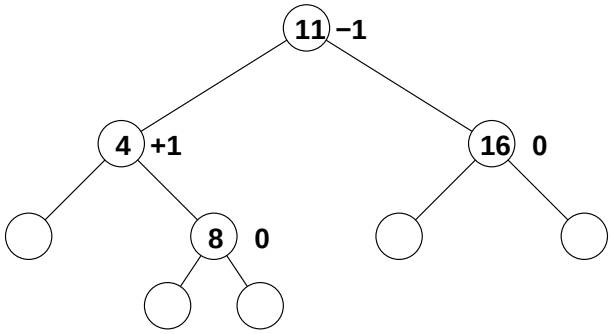
Eine Doppelrotation, d.h., zunächst eine Rotation nach links bei p und dann eine Rotation nach rechts bei $\varphi(p)$, stellt die AVL-Ausgeglichenheit bei $\varphi(p)$ wieder her.

Da nach der Doppelrotation der Teilbaum mit Wurzel $\varphi(p)$ nicht um eins gewachsen ist, ist eine weitere Umstrukturierung nicht notwendig.

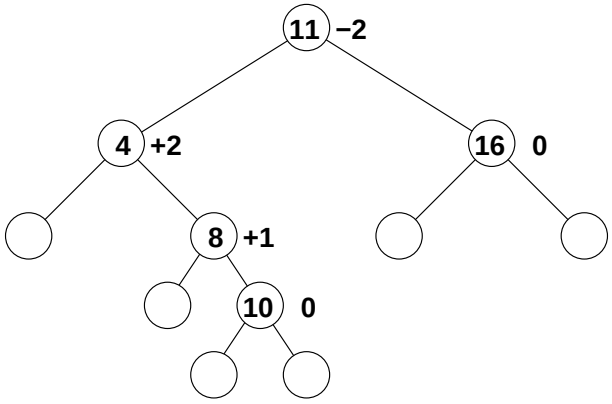
Der Fall $bal(p) = 0$ ist wegen der Invariante nicht möglich.

Im Fall 2 (p ist rechter Sohn seines Vaters $\varphi(p)$) geht man völlig analog vor.

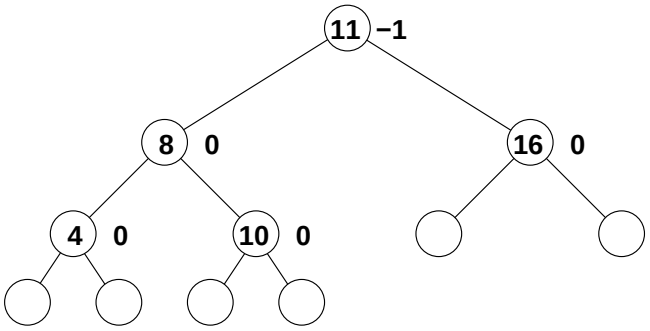
Beispiel:



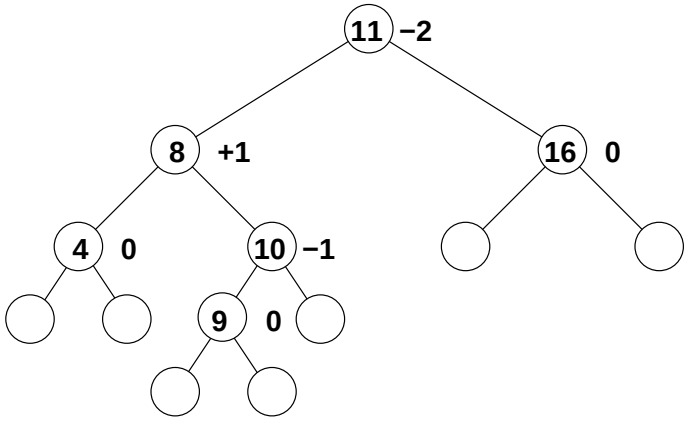
Füge Schlüssel 10 ein:



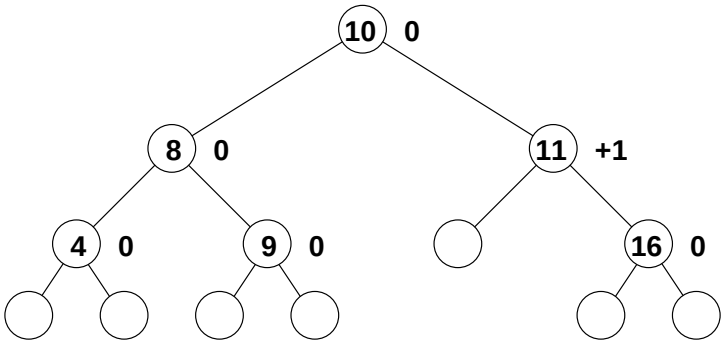
Eine Rotation nach links stellt den Ausgleich wieder her.



Füge Schlüssel 9 ein.



Eine Doppelrotation stellt den Ausgleich wieder her.



Das Einfügen eines Schlüssels in einen AVL-Baum mit N Schlüsseln ist in $O(\log(N))$ Schritten ausführbar.

Entfernen:

Umstrukturierungen werden mit einer Funktion `upout()` durchgeführt.

Zuerst sucht man den zu entfernenden Schlüssel.

Fall 1: Der zu entfernende Schlüssel ist der Schlüssel eines Knotens, dessen beide Söhne Blätter sind.

Entferne den Knoten und ersetze ihn durch ein Blatt.

Sei p der Vater des neuen Blattes, falls vorhanden.

Da der Teilbaum von p, der durch das Blatt ersetzt wurde, die Höhe 1 hatte, so muss der andere Teilbaum von p die Höhe 0, 1 oder 2 haben.

Hat er die Höhe 1, so ändert man die Balance von p von 0 auf +1 oder -1.

Hat er die Höhe 0, so ändert man die Balance von p von +1 oder -1 auf 0.

In diesem Fall ist die Höhe von p um eins gefallen, und die Vorgänger von p zurück zur Wurzel müssen ebenfalls mit `upout()` geprüft werden.

Hat der Teilbaum mit Wurzel p die Höhe 2, so führt man eine Umstrukturierung durch, indem `upout()` auf dem eingefügten Blatt ausgeführt wird.

Fall 2: Der zu entfernende Schlüssel ist der Schlüssel eines Knotens p, der nur einen inneren Knoten q als Sohn hat.

Dann müssen die beiden Söhne von q Blätter sein.

Ersetze den Schlüssel von p durch den Schlüssel von q und ersetze q durch ein Blatt.

Rufe `upout()` auf, um die Ausgeglichenheit wieder herzustellen.

Fall 3: Der zu entfernende Schlüssel ist der Schlüssel eines Knotens p, dessen beide Söhne innere Knoten sind.

Ersetze den Schlüssel durch den Schlüssel des symmetrischen Nachfolgers oder des symmetrischen Vorgängers und entferne den symmetrischen Nachfolger bzw. den symmetrischen Vorgänger (Fall 1 oder Fall 2).

Die Funktion `upout()`:

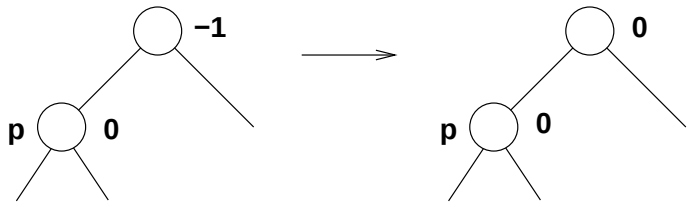
Invariante:

Wenn `upout(p)` aufgerufen wird, gilt $bal(p) = 0$ und der Teilbaum mit Wurzel p ist in der Höhe um 1 gefallen.

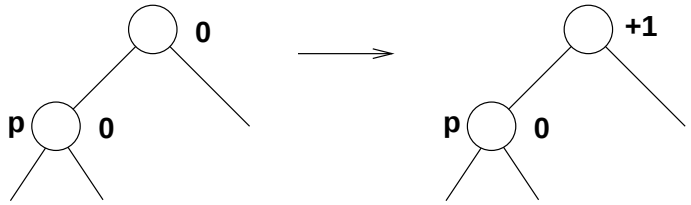
Wir unterscheiden wieder zwei Fälle, je nachdem ob p linker oder rechter Sohn seines Vaters $\varphi(p)$ ist.

Fall 1: (p ist linker Sohn seines Vaters $\varphi(p)$)

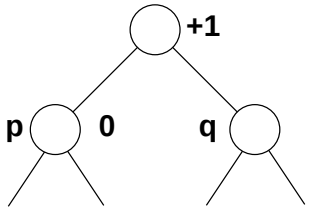
Fall 1.1: ($bal(\varphi(p)) = -1$)



Fall 1.2: ($bal(\varphi(p)) = 0$)

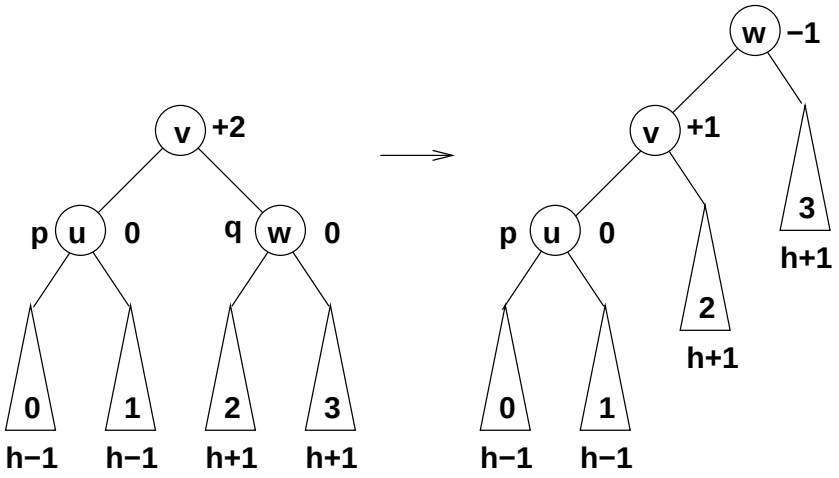


Fall 1.3: $(bal(\varphi(p)) = +1)$



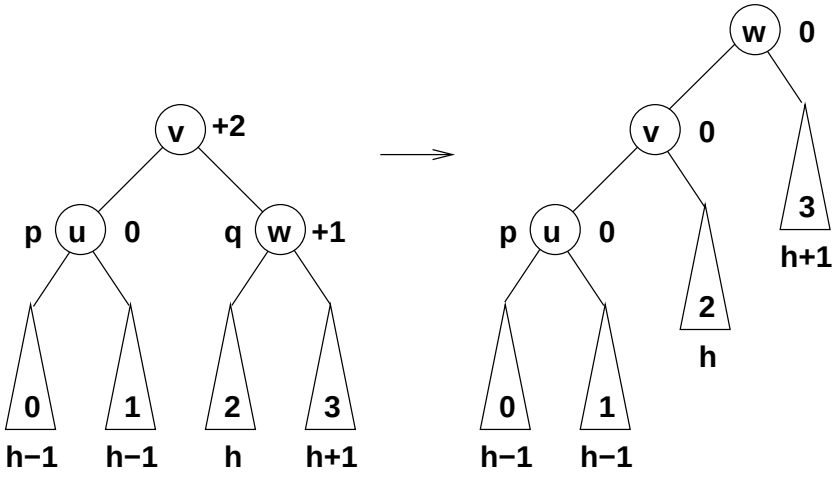
Der rechte Teilbaum von $\varphi(p)$ mit Wurzel q ist also höher als der linke mit Wurzel p, der darüberhinaus noch in der Höhe um 1 gefallen ist. Wir machen eine Fallunterscheidung nach dem Balancefaktor von q.

Fall 1.3.1: $(bal(q) = 0)$



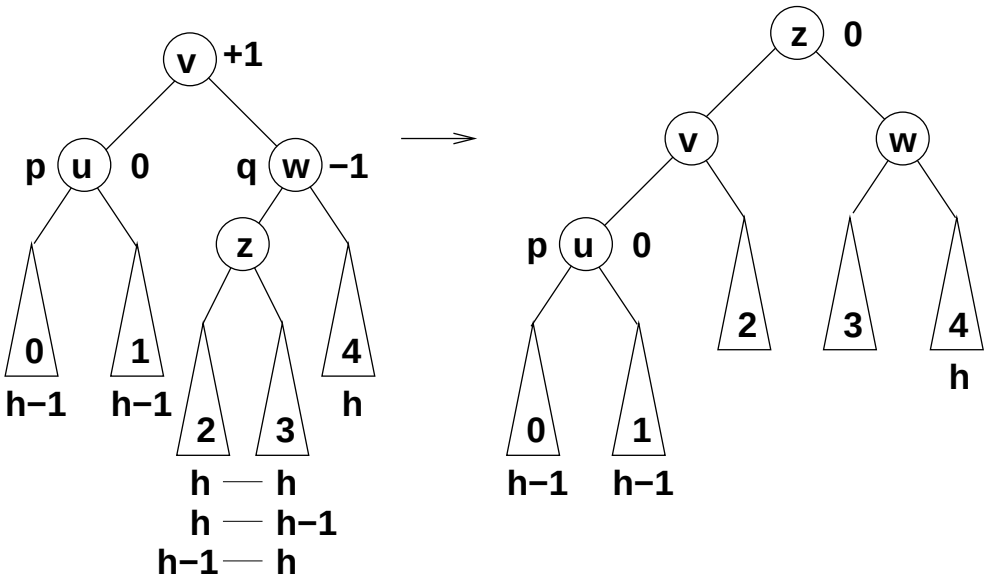
Eine Rotation nach links stellt die Ausgeglichenheit wieder her.

Fall 1.3.2: $(bal(q) = +1)$



Eine Rotation nach links stellt die Ausgeglichenheit wieder her.

Fall 1.3.3: $(bal(q) = -1)$



Eine Doppelrotation rechts-links stellt die Ausgeglichenheit wieder her.

Fall 2: (p ist rechter Sohn seines Vaters) ist völlig symmetrisch zum Fall 1.

Es kann also vorkommen, dass nach einer Rotation oder Doppelrotation die Funktion `upout()` erneut aufgerufen werden muss.

Da der Aufwand der Rotationen konstant ist und die Höhe des Baumes aus $O(\log(N))$ ist, kann ein Schlüssel in $O(\log(N))$ Schritten entfernt werden.

Die Anzahl der Rotationen gemittelt über die Anzahl der Operationen ist für AVL-Bäume pro Operation konstant. (Amortisierte Laufzeitanalyse.)

4.3 Splay-Bäume

Man möchte ohne explizite Speicherung von Balance-Informationen oder Häufigkeitszählern eine Strukturanpassung an unterschiedliche Zugriffshäufigkeiten erreichen.

Schlüssel, auf die relativ häufig zugegriffen wird, sollen näher zur Wurzel wandern. Dafür können Schlüssel, auf die seltener zugegriffen wird, zu den Blättern hinabwandern.

Die Zugriffshäufigkeiten sind vorher nicht bekannt.

Splay-Bäume sind selbstanordnende Suchbäume, die jeden Schlüssel x, auf den zugegriffen wurde, mit Umstrukturierungen zur Wurzel bewegt. Zugleich wird erreicht, dass sich die Längen sämtlicher Pfade zu Schlüsseln auf dem Suchpfad zu x etwa halbieren.

Die Splay-Operation:

Sei T ein Suchbaum und x ein Schlüssel. Dann ist splay(T, x) der Suchbaum, den man wie folgt erhält:

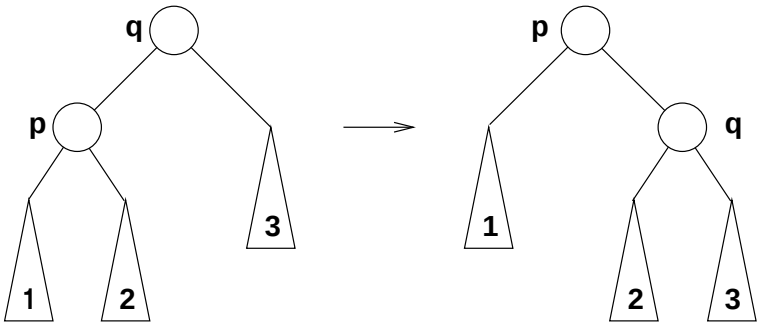
Schritt 1: Suche nach x in T. Sei p der Knoten, bei dem die (erfolgreiche) Suche endet, falls x in T vorkommt.

Ansonsten sei p der Vater des Blattes an dem die Suche nach x endet, falls x nicht in T vorkommt.

Schritt 2: Wiederhole die folgenden Operationen zig, zig-zig und zig-zag beginnend bei p solange, bis sie nicht mehr ausführbar sind, weil p Wurzel geworden ist.

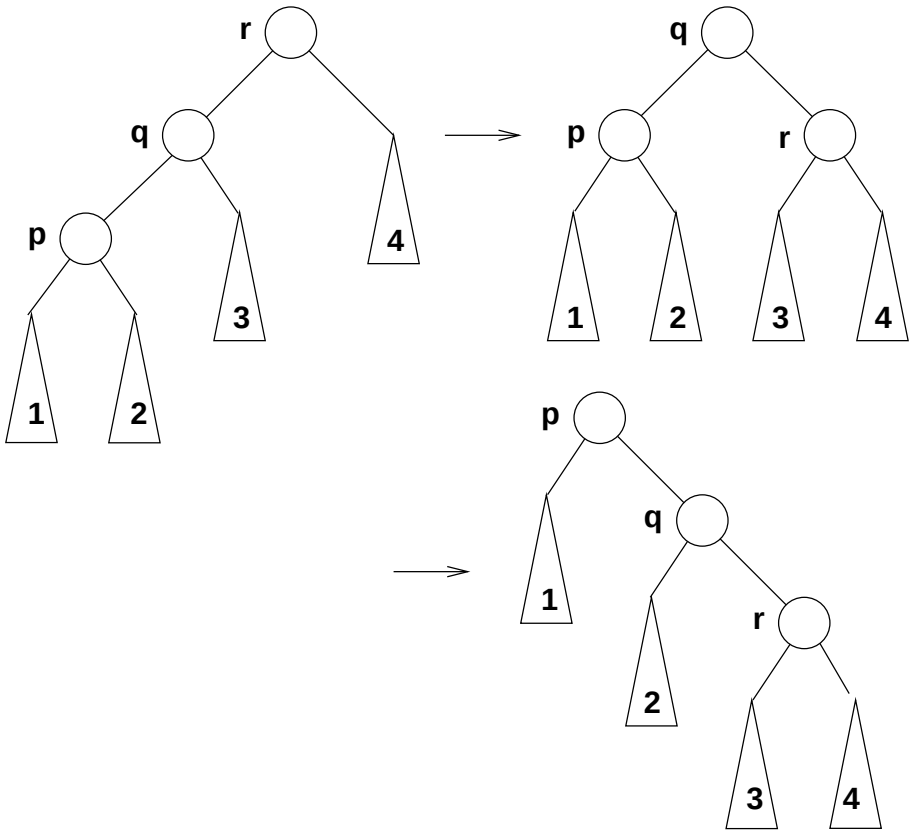
Fall 1: (p hat den Vater $q = \varphi(p)$ und $\varphi(p)$ ist die Wurzel)

Dann führe die Operation zig aus, d.h. eine Rotation nach links oder rechts, die p zur Wurzel macht.



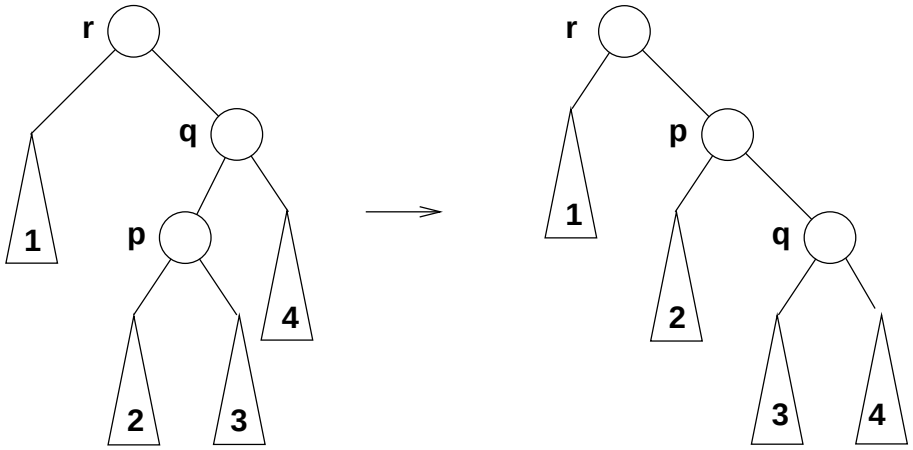
Fall 2: (p hat den Vater $q = \varphi(p)$ und den Großvater $r = \varphi(\varphi(p))$ und p sowie $\varphi(p)$ sind beides rechte oder beides linke Söhne)

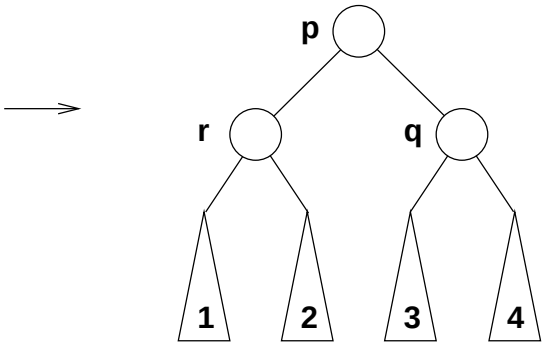
Dann führe die Operation zig-zig aus, d.h. zwei aufeinander folgende Rotationen in dieselbe Richtung, die p zwei Niveaustufen hinaufbewegen.



Fall 3: (p hat Vater $q = \varphi(p)$ und Großvater $r = \varphi(\varphi(p))$ und p ist linker Sohn von q und q ist rechter Sohn von r bzw. p ist rechter Sohn von q und q ist linker Sohn von r.

Dann führe die Operation zig-zag aus, d.h. zwei Rotationen in entgegengesetzter Richtung, die p zwei Niveaustufen hinaufbewegen.





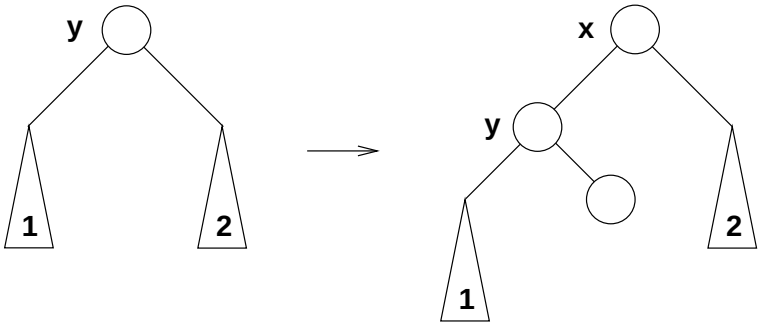
Kommt x in T vor, so erzeugt splay(T, x) einen Suchbaum, der Schlüssel x in der Wurzel speichert.

Kommt x nicht in T vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel x unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

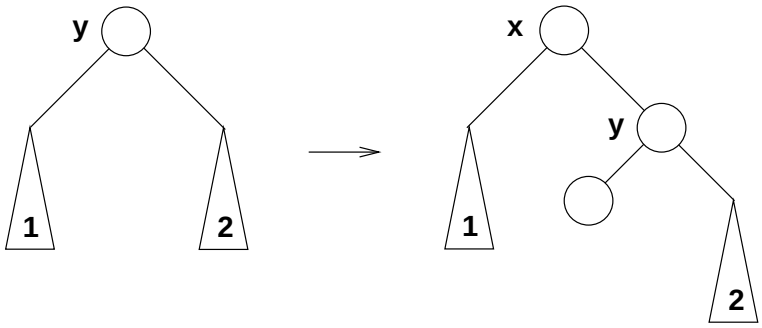
Suchen: Um nach x in T zu suchen wird splay(T, x) ausgeführt und x an der Wurzel gesucht.

Einfügen: Um x in T einzufügen, rufe splay(T, x) auf. Ist x nicht in der Wurzel, so füge wie folgt eine neue Wurzel mit x ein.

Falls der Schlüssel der Wurzel von T kleiner als x ist:



Falls der Schlüssel der Wurzel von T größer als x ist:



Entfernen: Um x aus T zu entfernen, rufe splay(T, x) auf.

Ist x in der Wurzel, so sei T_l der linke Teilbaum und T_r der rechte Teilbaum an der Wurzel.

Rufe splay(T_l, ∞) auf. Dadurch entsteht ein Teilbaum mit dem größten Schlüssel von T_l an der Wurzel und einen leeren rechten Teilbaum. Ersetze den rechten leeren Teilbaum durch T_r.

Bemerkung: Die Ausführung der Funktion splay(T, x) schließt stets die Suche nach x ein.

Alle Suchbäume, die man mit den hier angegebenen Verfahren erhält, wenn man ausgehend vom anfangs leeren Baum eine beliebige Folge von Such-, Einfüge- und Entferne-Operationen ausführt, heißt die Klasse der Splay-Bäume.

Das Ausführen einer beliebigen Folge von m Such-, Einfüge- und Entferne-Operationen, in der höchstens N mal eingefügt wird und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $O(m \cdot log(N))$ Schritte. (Amortisierte Laufzeitanalyse!)

4.4 B-Bäume

B-Bäume sind balancierte Baumstrukturen, die auch für eine externe Speicherung auf Platten, CD-ROMs, usw. gut geeignet sind.

Das Laden eines Blockes aus einem Hintergrundspeicher benötigt heutzutage in der Regel bis zu 10000 mal mehr Zeit als die Suche nach einem Schlüssel im Hauptspeicher.

Das Ziel ist es, möglichst selten Blöcke aus dem Hintergrundspeicher nachladen zu müssen.

Es wird versucht die Höhe der Bäume sehr klein zu halten, indem mehrere Schlüssel an einem inneren Knoten gespeichert werden.

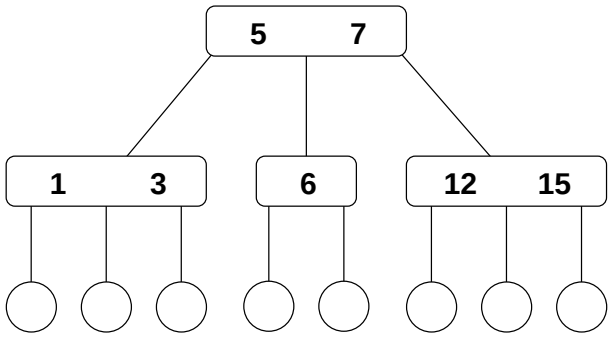
Ein B-Baum der Ordnung m, $m \geq 3$, ist ein Baum mit folgenden Eigenschaften:

1. Alle Blätter habe die gleiche Tiefe.
2. Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat mindestens $\lceil m/2 \rceil$ Söhne.
3. Die Wurzel hat mindestens 2 Söhne.
4. Jeder Knoten hat höchstens m Söhne.
5. Jeder Knoten mit i Söhnen speichert $i - 1$ Schlüssel.

Zusätzlich gilt:

Ist p ein innerer Knoten eines B-Baumes der Ordnung m, so speichert er l Schlüssel $s_1, \ldots, s_l$ und hat $l + 1$ Söhne $p_0, \ldots, p_l$ mit $\lceil \frac{m}{2} \rceil \leq l + 1 \leq m$.

Alle Schlüssel im Teilbaum mit Wurzel p_{i-1}, $1 \leq i \leq l$, sind kleiner als s_i und alle Schlüssel im Teilbaum mit Wurzel p_i sind größer als s_i.



Bemerkungen:

B-Bäume der Ordnung 3 heißen auch $2-3$-Bäume.

Obiger $2-3$-Baum speichert sieben Schlüssel und hat 8 Blätter.

Ein B-Baum, der N Schlüssel speichert, hat genau $N + 1$ Blätter.

Ein B-Baum der Ordnung m mit Höhe h hat die minimale Blattanzahl, wenn seine Wurzel nur 2 und jeder innere Knoten nur $\lceil m/2 \rceil$ viele Söhne hat.

Daher ist die minimale Blattanzahl

$$N_{\min} = 2 \cdot \lceil \frac{m}{2} \rceil^{h-1}.$$

Die Blattanzahl wird maximal, wenn jeder innere Knoten die maximal mögliche Anzahl von Söhnen hat.

Somit ist die maximale Blattanzahl

$$N_{\max} = m^h.$$

Also gilt:

$$N_{\min} = 2 \cdot \lceil \frac{m}{2} \rceil^{h-1} \leq N + 1 \leq m^h = N_{\max}$$

und somit

$$log_m(N+1) \quad \leq \quad h \quad \leq \quad 1 + \log_{\lceil \frac{m}{2} \rceil}(\frac{N+1}{2}).$$

B-Bäume sind also balanciert.

Die Ordnung der Bäume liegt in der Praxis üblicherweise bei etwa 100 bis 200.

Ist $m = 199$, so haben B-Bäume mit bis zu 1999999 Schlüsseln höchstens die Höhe 4.

Suchen:

Die Suche nach einem Schlüssel x in einem B-Baum T ist eine natürliche Verallgemeinerung der Suche in einem Suchbaum.

Man beginnt mit der Wurzel, die die Schlüssel $s_1, \ldots, s_l$ speichert, und bestimmt den kleinsten Index i mit $x \leq s_i$, falls es ein solches i gibt. Ansonsten ist x größer als der größte Schlüssel s_l.

Ist $x = s_i$, so ist x gefunden.

Ist $x < s_i$, so setzt man die Suche bei p_{i-1} fort; im zweiten Fall ($x > s_i$) setzt man die Suche bei p_i fort.

Das wird solange fortgesetzt, bis man x gefunden hat, oder an einem Blatt erfolglos endet.

Die Suche nach dem kleinsten Index i mit $x \leq s_i$ kann mit linearer Suche erfolgen.

Einfügen:

Um einen Schlüssel x in einem B-Baum einzufügen, sucht man zunächst nach x.

Sei p der Vater des Blattes, bei dem die erfolglose Suche von x in T endet. (Bei erfolgreicher Suche wird x nicht eingefügt.)

Knoten p habe die Schlüssel $s_1, \ldots, s_l$ gespeichert.

Angenommen die Suche endet im Sohn p_i von p.

Wir unterscheiden zwei Fälle:

Fall 1: p hat noch nicht die maximale Anzahl $m-1$ von Schlüsseln gespeichert. Dann fügen wir x in p zwischen s_i und s_{i+1} (bzw. vor s_1 falls $i = 0$ oder nach s_l falls $i = l$) ein, und schaffen ein neues Blatt.

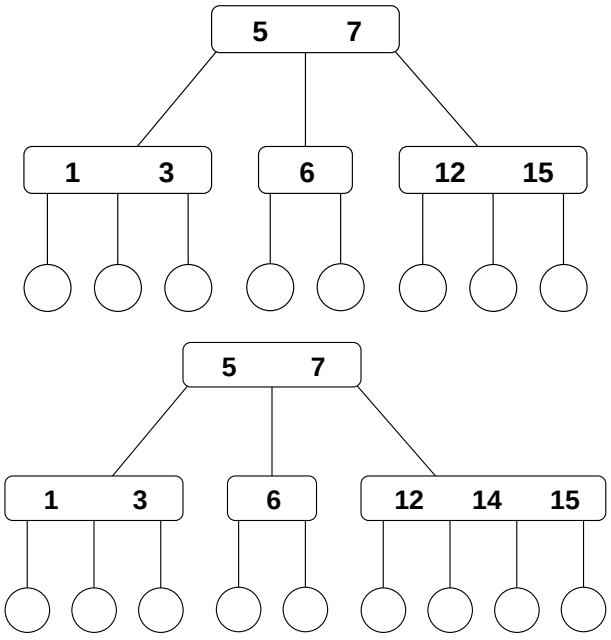
Fall 2: p hat bereits die maximale Anzahl $m-1$ von Schlüsseln gespeichert. In diesem Fall ordnen wir den Schlüssel x wie im Fall 1 entsprechend seiner Größe in p ein und teilen anschließend den zu groß gewordenen Knoten in der Mitte auf.

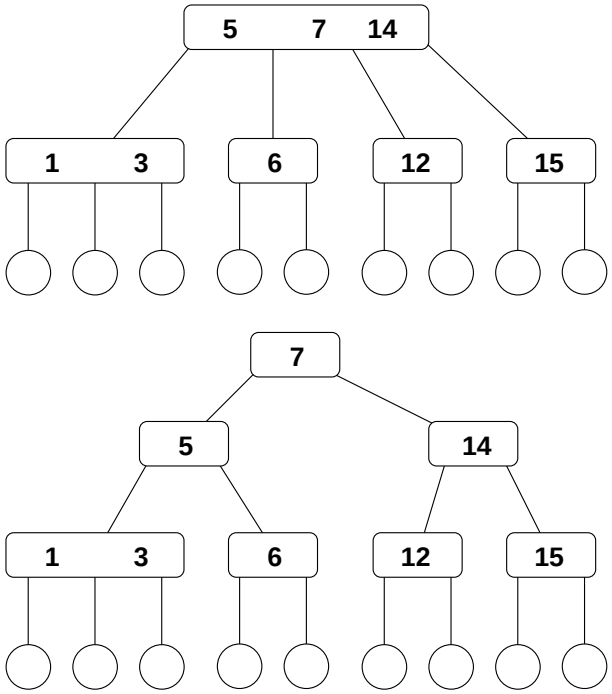
Sind $k_1, \ldots, k_m$ die Schlüssel in dem zu großen Knoten in aufsteigender Reihenfolge, so bildet man zwei neue Knoten mit den Schlüsseln $k_1, \ldots, k_{\lceil \frac{m}{2} \rceil - 1}$ und $k_{\lceil \frac{m}{2} \rceil + 1}, \ldots, k_m$ und fügt den mittleren Schlüssel $k_{\lceil \frac{m}{2} \rceil}$ auf dieselbe Weise im Vater des Knotens p ein.

Dieses Teilen wird solange längs des Suchpfades bis zur Wurzel fortgesetzt, bis ein Knoten erreicht ist, der noch nicht die maximale Anzahl von Schlüsseln speichert, oder bis die Wurzel erreicht wird.

Muss die Wurzel geteilt werden, so schafft man eine neue Wurzel, die den mittleren Schlüssel als einzigen Schlüssel speichert.

Beispiel: Füge Schlüssel 14 ein (für $m = 3$):





Entfernen:

Zum Entfernen eines Schlüssels x, wird x zunächst gesucht.

Fall 1: x ist Schlüssel s_i eines Knotens, und p_{i-1}, p_i sind keine Blätter.

Dann suche den größten Schlüssel y im Teilbaum mit Wurzel p_{i-1} bzw. den kleinsten Schlüssel y im Teilbaum mit Wurzel p_i und tausche ihn mit x. Entferne anschließend x wie im Fall 2.

Fall 2: x ist Schlüssel s_i eines Knotens und p_{i-1}, p_i sind Blätter.

Entferne x und das Blatt p_{i-1} oder p_i.

Falls der dabei entstandene Knoten p weniger als $\lceil \frac{m}{2} \rceil - 1$ Schlüssel speichert, sei q der linke oder rechte Bruder von p, je nachdem welcher Bruder existiert.

O.B.d.A. sei q linke Bruder von p.

Sei y der Schlüssel im Vater r, der zwischen q und p angeordnet ist.

Wir speichern nun die Schlüssel von q, den Schlüssel y, und die Schlüssel von p in einen neuen Knoten p'.

Dazu wird y aus dem Vater r entfernt.

Fall 2.1: Sind $k_1, \ldots, k_l$ die Schlüssel in p' in aufsteigender Reihenfolge und ist $l > m-1$, so bildet man zwei neue Knoten p und q mit den Schlüsseln $k_1, \ldots, k_{\lceil \frac{l}{2} \rceil - 1}$

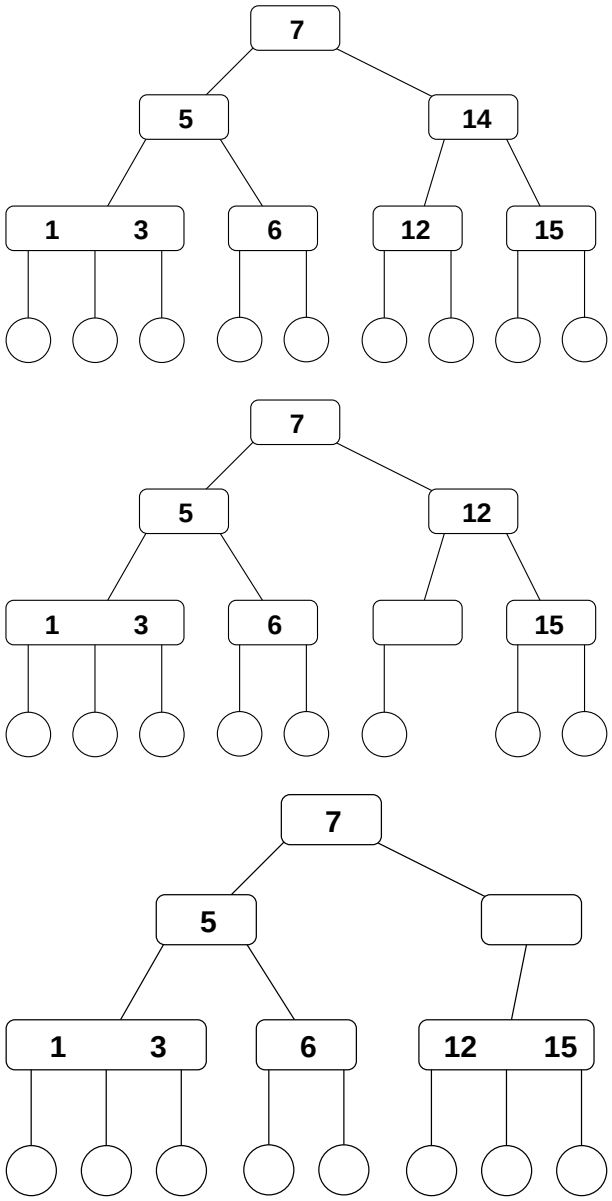
und $k_{\lceil \frac{l}{2} \rceil + 1}, \ldots, k_l$ und fügt den mittleren Schlüssel $k_{\lceil \frac{l}{2} \rceil}$ im Vater r an der entsprechenden Position zwischen q und p ein.

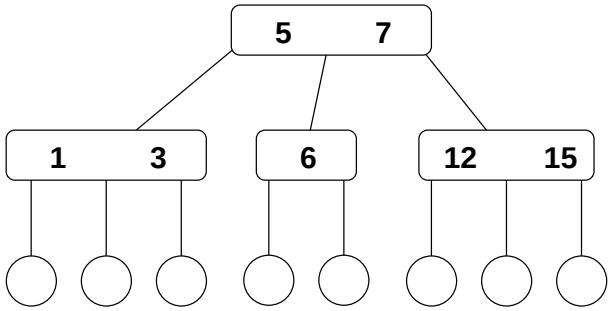
Fertig!

Fall 2.2: Ist $l \leq m - 1$, dann muss der Vater r von p' weiter betrachtet werden, weil er möglicherweise zu wenige Schlüssel speichert.

Die Korrekturen der zu klein gewordenen Knoten setzen sich entlang des Suchpfades gegebenenfalls bis zur Wurzel fort.

Beispiel: Entferne Schlüssel 14 (für $m = 3$):





Bemerkungen:

Man kann offenbar alle drei Operationen Suchen, Einfügen und Entfernen in $O(\log_{\lceil \frac{m}{2} \rceil}(N))$ Schritten durchführen (für m konstant).

Das Verhalten im Mittel ist jedoch besser (ohne Beweis).

Die Knoten werden in der Regel so groß gewählt, dass sie genau auf eine Seite des Hintergrundspeichers passen.

5 Suchen und Anordnen in Graphen

5.1 Notationen

Definition: Ein Graph $G = (V, E)$ besteht aus einer endlichen Menge von Knoten $V = \{u_1, \ldots, u_n\}$ und einer endlichen Menge E von Kanten.

1. In einem gerichteten Graphen ist jede Kante e ein Paar (u, v) von zwei Knoten $u, v \in V$, u ist der Startknoten und v der Zielknoten der Kante (u, v).

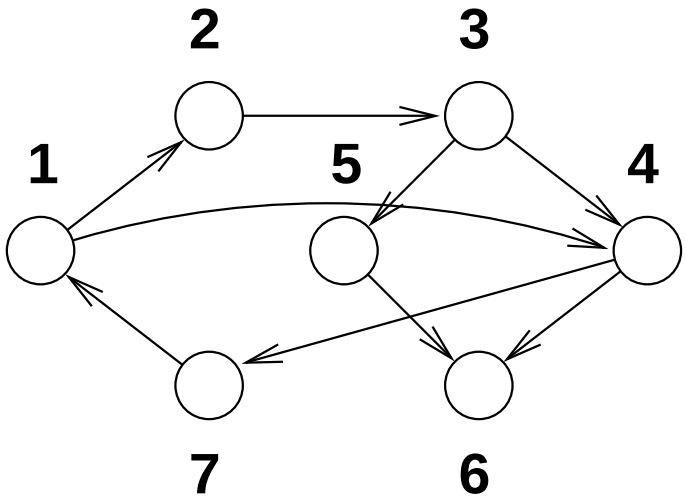
$$E \subseteq V \times V$$

2. In einem ungerichteten Graphen ist jede Kante e ein Menge $\{u, v\}$ von zwei verschiedenen Knoten $u, v \in V$, u und v sind die Endknoten der Kante $\{u, v\}$.

$$E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$$

Wir zeichnen die Knoten der Graphen als Kreise oder Punkte. Eine gerichtete Kante (u, v) zeichnen wir als Pfeil von u nach v. Eine ungerichtete Kante zeichnen wir als verbindende Linie zwischen u und v.

Beispiel:



Sei $e = (u, v)$ bzw. $e = \{u, v\}$ eine Kante. Die Knoten u und v sind zueinander adjazent und mit Kante e inzident.

Adjazenzmatrizen:

Sei n die Anzahl der Knoten in einem gerichteten Graphen $G = (V, E)$. Die Adjazenzmatrix für G ist eine $n \times n$-Matrix $A_G = (a_{i,j})$ mit

$$a_{i,j} = \begin{cases} 1 & \text{falls } (u_i, u_j) \in E \\ 0 & \text{falls } (u_i, u_j) \notin E \end{cases}$$

Für die Repräsentation von ungerichteten Graphen mit Adjazenzmatritzen wird jede ungerichteten Kante $e = \{u, v\}$ durch zwei gerichtete Kanten $(u, v), (v, u)$ gespeichert.

Sei $V = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7\}$ und
$E = \{(u_1, u_2), (u_1, u_4), (u_2, u_3), (u_3, u_4), (u_3, u_5), (u_4, u_6), (u_4, u_7), (u_5, u_6), (u_7, u_1)\}$,

dann ist die Adjazenzmatrix A_G für G:

$$A_G = \left\{ \begin{array}{c|ccccccc} & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 \\ \hline u_1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ u_2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ u_3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ u_4 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ u_5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ u_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ u_7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right\}$$

Die Repräsentation eines Graphens $G = (V, E)$ mit n Knoten und m Kanten als Adjazenzmatrix benötigt $\Theta(n^2)$ Platz.

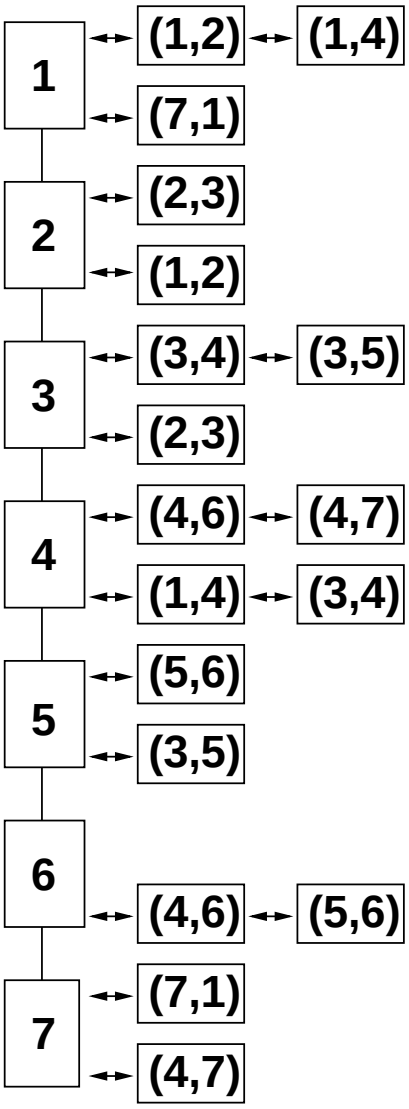
Typische Operationen wie etwa das Inspizieren aller mit einem Knoten inzidenten Kanten sind auf Adjazensmatrizen ineffizient.

Adjazenzlisten:

Für jeden Knoten u werden in einer doppelt verketteten Liste (Adjazenzliste) alle von u ausgehenden und einlaufenden Kanten bzw. mit u inzidenten Kanten gespeichert.

Die Darstellung eines Graphens G mit n Knoten und m Kanten als Adjazenzliste benötigt $\Theta(n + m)$ Platz.

Beispiel:



Adajzenzlisten unterstützen sehr gut das Verfolgen von Kanten (die Wegesuche).

Notationen für ungerichtete Graphen:
Sei $G = (V, E)$ ein ungerichteter Graph.

- Der Kotengrad eines Knotens u, geschrieben $\deg(u)$, ist die Anzahl der mit u inzidenten Kanten.
- Ein ungerichteter Graph $G' = (V', E')$ ist ein Teilgraph von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- $G' = (V', E') \subseteq G$ ist ein induzierter Teilgraph von G, falls $E' = E \cap \{\{u, v\} \mid u, v \in V'\}$. Für eine Knotenmenge $V' \subseteq V$ ist $G|_{V'} = (V', E \cap \{\{u, v\} \mid u, v \in V'\}$ der durch V' induzierte Teilgraph von G.
- Ein ungerichteter Weg p in G der Länge k von Knoten u nach Knoten w ist eine Folge von k Knoten

$$p = (v_1, \ldots, v_k)$$

mit $v_1 = u$ und $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 < i \leq k$.

Weg p ist einfach, wenn alle k Knoten $v_1, \ldots, v_k$ verschieden sind.

- Ein (einfacher) ungerichteter Kreis der Länge $k \geq 3$ ist ein (einfacher) ungerichteter Weg $p = (v_1, \ldots, v_k)$ der Länge k mit $\{v_k, v_1\} \in E$.
- Ein ungerichteter Wald ist ein ungerichteter, kreisfreier Graph.
- Ein ungerichteter Wald in dem es zwischen jedem Knotenpaar einen Weg gibt, ist ein ungerichteter Baum.

  In einem ungerichteten Baum gibt es zwischen jedem Knotenpaar genau einen einfachen Weg.

Notationen für gerichtete Graphen: Sei $G = (V, E)$ ein gerichteter Graph.

- Der Eingangsgrad bzw. Ausgangsgrad eines Knotens u, geschrieben $\text{indeg}(u)$ bzw. $\text{outdeg}(u)$, ist die Anzahl der in u einlaufenden bzw. auslaufenden Kanten.
- Der Grad eines Knotens u, geschrieben $\deg(u)$, ist die Anzahl der mit u inzidenten Kanten ($\deg(u) = \text{indeg}(u) + \text{outdeg}(u)$).
- Ein gerichteter Graph $G' = (V', E')$ ist ein Teilgraph von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- $G' = (V', E') \subseteq G$ ist ein induzierter Teilgraph von G, falls $E' = E \cap (V' \times V')$. Für eine Knotenmenge $V' \subseteq V$ ist $G|_{V'} = (V', E \cap (V' \times V'))$ der durch V' induzierte Teilgraph von G.
- Ein gerichteter Weg p in G der Länge k von Knoten u nach Knoten w ist eine Folge von k Knoten

  $$p = (v_1, \ldots, v_k)$$

  mit $v_1 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 < i \leq k$.

  Weg p ist einfach, wenn alle Knoten $v_1, \ldots, v_k$ verschieden sind.
- Ein (einfacher) gerichteter Kreis der Länge $k \geq 1$ ist ein (einfacher) gerichteter Weg $p = (v_1, \ldots, v_k)$ der Länge k mit $(v_k, v_1) \in E$.

  Ein Graph heißt kreisfrei, wenn er keinen Kreis enthält.
- Ein gerichteter Wald ist ein gerichteter, kreisfreier Graph mit $\text{indeg}(u) \leq 1$ für alle Knoten u.

  Die Knoten mit Eingangsgrad 0 heißen Wurzeln.

- Ein gerichteter Wald mit genau einer Wurzel ist ein gerichteter Baum.

In einem gerichteten Baum gibt es von der Wurzel zu jedem Knoten genau einen Weg.

Zwei Graphen $G = (V, E)$ und $J = (V', E')$ sind isomorph, falls es eine Bijektion

$$b : V \longrightarrow V'$$

gibt mit

$(u, v) \in E \Longleftrightarrow (b(u), b(v)) \in E'$, falls G und J gerichtet sind, bzw.
$\{u, v\} \in E \Longleftrightarrow \{b(u), b(v)\} \in E'$, falls G und J ungerichtet sind.

Graphisomorphie nachzuweisen ist im allgemeinen schwer (vermutlich jedoch nicht NP-vollständig). Die besten Algorithmen (Stand: 1995) haben eine Laufzeit von $O(n^{\log(n)})$.

Satz: (ohne Beweis) Sei P eine Grapheigenschaft, d.h., $P(G)$ ist wahr oder falsch. Angenommen P besitzt die folgenden zusätzlichen Eigenschaften:

- P ist nicht trivial, d.h., es gibt mindestens einen Graphen, der P erfüllt, und mindestens einen Graphen, der P nicht erfüllt.
- Für isomorphe Graphen G und J gilt immer $P(G) = P(J)$.
- P ist monoton, d.h., wenn J Teilgraph von G ist, dann folgt aus $P(G)$ auch $P(J)$.

Dann benötigt jeder Algorithmus, der P auf der Basis einer Adjazenzmatrix für G entscheidet, mindestens $\Omega(n^2)$ Schritte.

Beispiele: Planarität, Kreisfreiheit, ...

5.2 Topologische Sortierung

Eine topologische Sortierung eines gerichteten Graphens $G = (V, E)$ mit n Knoten ist eine Anordnung der Knoten

$$h : V \to \{1, \ldots, n\},$$

so dass für jede gerichtete Kante $(u_i, u_j) \in E$, $h(u_i) < h(u_j)$.

Satz: Ein gerichteter Graph besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist.

Beweis: Aus der Existenz einer topologischen Sortierung folgt unmittelbar dass G kreisfrei ist.

Die Existenz einer topolgischen Sortierung für kreisfreie Graphen folgt aus folgender induktiven Überlegung.

Für kreisfreie Graphen mit genau einem Knoten u ist $h(u) = 1$ eine topologische Sortierung.

Sei nun $n > 1$ und u ein Knoten mit $\text{indeg}(u) = 0$. Ein solcher Knoten existiert immer, wenn G kreisfrei ist. Er kann wie folgt gefunden werden: Starte bei einem beliebigen Knoten u und verfolge die Kanten rückwärts bis man zu einem Knoten kommt, der keine einlaufende Kanten hat.

Eine topologische Sortierung für G enthält man, wenn man an der topologischen Sortierung für G ohne u und seine inzidenten Kanten, den Knoten u vorne anfügt. □.

Algorithmus zur topologischen Sortierung eines gerichteten Graphens:

Pseudosprache:

Speichere an jedem Knoten u eine Zahl $I[u]$;

Initialisiere $I[u]$ für alle Knoten u mit $\text{indeg}(u)$;

Speichere in einer Liste L alle Knoten u mit $I[u] = 0$;

Solange L noch nicht leer ist, wiederhole {

Entnehme einen Knoten u aus L;

Füge u an die bis jetzt berechnete Folge hinten an;

Betrachte alle aus u auslaufenden Kanten (u, v) und erniedrige jedes $I[v]$ um 1, falls $I[v]$ dabei 0 wird, füge v zu L hinzu; }

Laufzeitanalyse:

Laufzeit: $O(|V| + |E|)$.

5.3 Transitiver Abschluss

Ein gerichteter Graph $G' = (V', E')$ ist der transitive Abschluss eines gerichteten Graphens $G = (V, E)$, wenn $V' = V$ und für alle Knoten $u, v \in V$ gilt, in G' gibt es genau dann eine Kante (u, v), wenn es in G einen Weg von u nach v gibt.

Sei $V = \{u_1, \ldots, u_n\}$ die Kontenmenge eines Graphens $G = (V, E)$. Jeder einfache Weg p von u_i nach u_j der Länge ≥ 3 kann in zwei einfache Wege p_1 von u_i nach u_k und p_2 von u_k nach u_j zerlegt werden, wobei Knoten u_k der Knoten mit dem größten Index k auf dem Weg p zwischen u_i und u_j ist.

Wir betrachten Wege in einer Reihenfolge, die sicherstellt, dass beim Zusammensetzen der Wege p_1 und p_2 beide Wege nur Knoten mit einem Index kleiner als k besuchen.

Invariante: Für jedes k wurden bereits alle Wege betrachtet, die nur Zwischenknoten mit einem Index kleiner als k besitzen.

Algorithmus zur Berechnung des transitiven Abschlusses:
Sei n die Anzahl der Knoten in G und A die Adjazenzmatrix.

```
for (i := 1, i ≤ n; i++) { A[i,i] := 1; }
for (k := 1; k ≤ n; k++)
    for (i := 1; i ≤ n; i++)
        for (j := 1; j ≤ n; j++)
            if (A[i,k] = 1 und A[k,j] = 1) then { A[i,j] := 1; }
```

Laufzeitanalyse:

Laufzeit: $O(|V|^3)$.

Einfache Verbesserung:

```
for (i := 1, i ≤ n; i++) { A[i,i] := 1; }
for (k := 1; k ≤ n; k++)
    for (i := 1; i ≤ n; i++)
        if (A[i,k] = 1) then
            for (j := 1; j ≤ n; j++)
              if (A[k,j] = 1) then { A[i,j] := 1; }
```

Nun wird die innere for-Schleife nicht $O(|V|^2)$ mal durchlaufen, sondern nur dann, wenn es eine Kante von u_i nach u_k gibt. Wenn m^* die Anzahl der Kanten im transitiven Abschluss von G ist, dann hat obiger Algorithmus eine Laufzeit von $O(|V|^2 + m^* \cdot |V|)$.

5.4 Tiefensuche und Breitensuche.

Die Suche soll in einem gerichteten Graphen alle von einem Startknoten s erreichbaren Knoten finden:

```
Suche(s)
{
    Markiere alle Knoten mit „unbesucht";
    Markiere den Startknoten s mit „besucht";
    Füge alle aus s auslaufenden Kanten in eine Datenstruktur L ein;
    Solange L nicht leer ist {
        Entnehme eine Kante (u,v) aus L;
        Falls Knoten v mit „unbesucht" markiert ist {
            Markiere Knoten v mit besucht;
            Füge alle aus v auslaufenden Kanten in L ein; } }
}
```

In der Datenstruktur L speichern wir diejenigen Kanten, von denen vielleicht noch unbesuchte Knoten erreicht werden können.

Jede Kante wird höchstens einmal in L eingefügt.

Jeder Knoten wird höchstens einmal inspiziert.

Die benötigte Zeit ist proportional zur Summe der Anzahlen der vom Startknoten erreichbaren Knoten und Kanten, also $O(|V| + |E|)$, falls das Einfügen in L und Entfernen aus L in konstanter Zeit möglich ist.

Die Suche funktioniert mit leichter Modifikation natürlich auch für ungereichtete Graphen.

Suche(s)
{

Markiere alle Knoten mit „unbesucht";

Markiere den Startknoten s mit „besucht";

Füge alle mit s inzidenten Kanten in eine Datenstruktur L ein;

Solange L nicht leer ist {

Entnehme eine Kante $\{u, v\}$ aus L;

Falls Knoten u bzw. v mit „unbesucht" markiert ist {

Markiere Knoten u bzw. v mit „besucht";

Füge alle mit u bzw. v inzidenten Kanten in L ein, falls sie noch nicht drin sind; } }

}

Der Typ der Datenstruktur L bestimmt die Durchlaufordnung.

Ist L ein Stack (first in, last out), dann werden die Knoten in Tiefensuche besucht.

Ist L eine Liste (first in, first out), dann werden die Knoten in Breitensuche besucht.

Werden bei der Tiefensuche die Knoten in der Reihenfolge nummerieren, in der sie markiert werden, erhält man eine (depth-first-search) DFS-Nummerierung.

Die Tiefensuch kann einfach rekursiv implementiert werden. Dadurch erspart man sich die Implementierung eines Stacks.

Definiert werden ein DFS-Index und ein DFE-Index (DFS-End-Index).

Rekursive Tiefensuche :

Markiere alle Knoten mit „unbesucht";

Zähler := 1;

Endzähler := 1;

für jeden Knoten $u \in V$ {

Tiefensuche(s); }

Tiefensuche(u)
{

Markiere u mit „besucht";

DFS[u] := Zähler + +;

Betrachte alle Kanten $(u, v) \in E$ {

Falls v mit „unbesucht" markiert ist {
Tiefensuche(v); } }

DFE[u] := Endzähler + +;

}

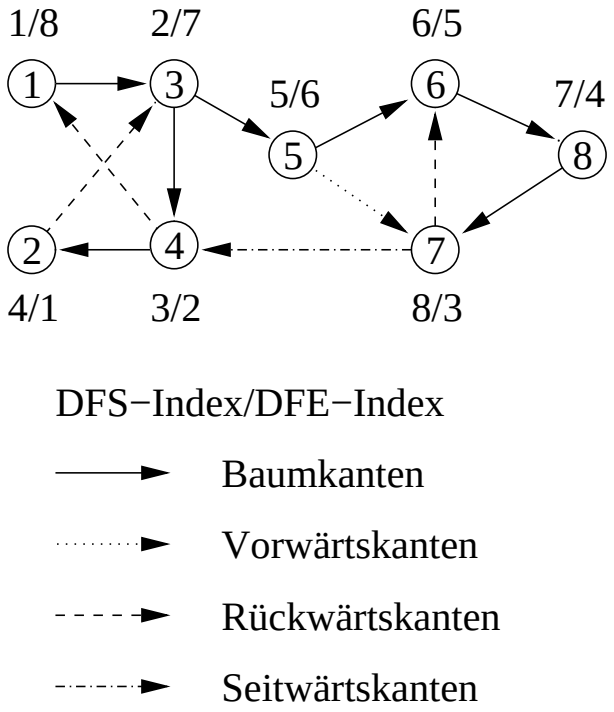
Wir unterscheiden die Kanten bei einem gerichteten Graphen nach der Rolle, die sie bei der Tiefensuche spielen.

Kanten, denen die Tiefensuche folgt, heißen Baumkanten.

Kanten (u, v) mit DFS[v] > DFS[u], die keine Baumkanten sind, heißen Vorwärtskanten.

Kanten (u, v) mit DFS[v] < DFS[u] und DFE[v] < DFE[u] heißen Querkanten oder Seitwärtskanten.

Kanten (u, v) mit DFS[v] < DFS[u] und DFE[v] > DFE[u] heißen Rückwärtskanten.



Die Baumkanten bilden einen Tiefensuchewald. Die Bäume im Tiefensuchewald heißen Tiefensuchebäume.

5.5 Zusammenhangsprobleme

Definitionen:

Ein ungerichteter Graph G ist zusammenhängend bzw. einfach zusammenhängend, wenn es zwischen jedem Knotenpaar in G einen ungerichteten Weg gibt.

Er ist zweifach (k-fach) zusammenhängend, wenn es zwischen jedem Knotenpaar zwei (bzw. k) knotendisjunkte Wege gibt. (Zwei Wege sind knotendisjunkt, wenn alle Knoten auf den beiden Wegen (bis auf Start- und Endknoten) paarweise verschieden sind).

Eine k-fache Zusammenhangskomponente von G ist ein maximaler (bzgl. der Knotenmenge) k-fach zusammenhängender, induzierter Teilgraph von G.

Ein gerichteter Graph G ist stark zusammenhängend, wenn es zwischen jedem Knotenpaar in G einen gerichteten Weg gibt.

Eine starke Zusammenhangskomponente von G ist ein maximaler (bzgl. der Knotenmenge), stark zusammenhängender, induzierter Teilgraph von G.

Für einen gerichteten Graphen $G = (V, E)$ sei $G' = (V, \{\{u, v\} \mid (u, v) \in E \vee (v, u) \in E\})$ der unterliegende ungerichtete Graph. Ein gerichteter Graph

G ist schwach zusammenhängend, wenn der unterliegende ungerichtete Graph G' zusammenhängend ist.

Zusammenhang und schwacher Zusammenhang sind Eigenschaften, die für ungerichtete bzw. gerichtete Graphen einfach mit einer Tiefensuche in linearer Zeit ($O(|V| + |E|)$) überprüft werden können.

Satz:

Ein ungerichteter Graph mit mindestens $k + 1$ Knoten ist genau dann k-fach zusammenhängend, wenn er nicht durch Entfernen von $k - 1$ Knoten unzusammenhängend werden kann.

Berechnung der zweifachen Zusammenhangskomponenten:

Die zweifachen Zusammenhangskomponenten in einem Graphen sind nicht immer paarweise knotendisjunkt, aber dafür immer kantendisjunkt. Daher spezifizieren wir eine zweifache Zusammenhangskomponente eindeutig durch die Menge ihrer Kanten.

Ein Knoten u ist ein Schnittpunkt bzw. Artikulationspunkt, wenn der Graph G ohne u aus mehr Zusammenhangskomponenten als G besteht.

Zur Berechnung der zweifachen Zusammenhangskomponenten ermitteln wir die Schnittpunkte mit Hilfe der folgenden Überlegungen.

1. Beim Durchlaufen eines ungerichteten Graphens gibt es keine Querkanten.
2. Trifft man während der Tiefensuche auf einen Schnittpunkt v, so muss sich mindestens eine 2-fache Zusammenhangskomponente in einem Teilbaum mit Wurzel v des Tiefensuchebaumes befinden; aus einem solchen Teilbaum heraus gibt es keine Rückwärtskante zu einem Vorgänger von v.
3. Wenn ein Schnittpunkt v Wurzel eines Tiefensuchebaumes ist, so hat v im Tiefensuchebaum mehr als einen Sohn, weil die Tiefensuche nur über v von einer Zusammenhangskomponente in die andere gelangen kann.

Wir merken uns in einer Variablen $P[v]$ wie weit man über Rückwärtskanten höchstens im DFS-Index zurückgelangen kann. Ist $P[v] \geq$ DFS$[v]$, dann ist v ein Schnittpunkt.

Markiere alle Knoten als „unbesucht";

Zähler := 1;

Initialisiere einen leeren Stack;

Für alle Knoten $u \in V$ {

Falls u mit „unbesucht" markiert ist, dann 2ZSuche(u); }

2ZSuche(u)
{

Markiere u als „besucht";

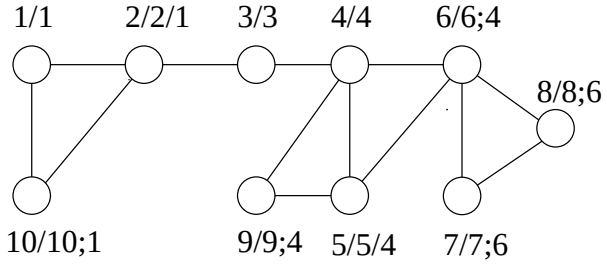
DFS[u] := Zähler + +;

$P[u]$:= DFS[u];

Betrachte alle mit u inzidenten Kanten $\{u, v\} \in E$ {

Lege $\{u, v\}$ auf den Stack, falls noch nicht geschehen;
Falls v mit „unbesucht" markiert ist {
Vater[v] := u;
2ZSuche(v);
Falls $P[v] \geq$ DFS[u], dann nimm alle Kanten bis einschließlich $\{u, v\}$ vom Stack und gebe sie als eine Komponente aus;
$P[u] := \min(P[u], P[v])$; }
Sonst {
Falls $v \neq$ Vater[u], dann $P[u] := \min(P[u],$ DFS[v]); } }

}



Bemerkungen:

1. Eine ungerichtete Kante wird in der Tiefensuche in beiden Richtungen inspiziert. Die kleinste erreichbare DFS-Nummer darf jedoch nicht über eine entgegengesetzte Baumkante ermittelt werden. Dies sind keine Rückwärtskanten. Ob eine Baumkante in entgegengesetzter Richtung betrachtet wird, kann mit Hilfe des gespeicherten Vaters festgestellt werden.

2. Damit auch die Kanten vom Stapel entfernt werden, die zu keiner zweifachen Zusammenhangskomponente gehören (die sogenannten Brücken), werden auch dann Kanten vom Stack entfernt, wenn $P[v] > \text{DFS}[u]$ und nicht nur wenn $P[v] = \text{DFS}[u]$ gilt.

3. Obiger Algorithmus hat offensichtlich eine Laufzeit von $O(|V| + |E|)$.

Algorithmus zur Berechnung der starken Zusammenhangskomponenten:

Die starken Zusammenhangskomponenten sind knotendisjunkt. Somit können wir die starken Zusammenhangskomponenten durch ihre Knotenmengen spezifizieren.

Die Berechnung der starken Zusammenhangskomponenten erfolgt analog zur Berechnung der zweifachen Zusammenhangskomponenten.

Markiere alle Knoten als „unbesucht";

Zähler := 1;

Initialisiere einen leeren Stack;

Für alle Knoten $u \in V$ {

Falls u mit „unbesucht" markiert ist, dann SZSuche(u); }

SZSuche(u)
{

Markiere u als „besucht";

DFS[u] := Q[u] := Zähler++;

Lege u auf den Stack;

Betrachte alle aus u auslaufenden Kanten $(u, v) \in E$ {

Falls v mit „unbesucht" markiert ist {
SZSuche(v);
Q[u] := min(Q[u], Q[v]); }
Sonst {
Falls DFS[v] < DFS[u] und v auf dem Stack liegt, dann Q[u] := min(Q[u], DFS[v]); } }

Falls $Q[u] = \text{DFS}[u]$, dann nimm alle Knoten bis einschließlich u vom Stack und gebe sie als eine Komponente aus;

}

Die Berechnung der starken Zusammenhangskomponenten benötigt offensichtlich $O(|V| + |E|)$ Schritte.

Graphen mit wenigen starken Zusammenhangskomponenten sind oft effizienter zu analysieren als Graphen mit vielen starken Zusammenhangskomponenten.

Beispiel: Transitiver Abschluss.

Berechnung des transitiven Abschlusses für einen gerichteten Graphen $G = (V, E)$ mit k starken Zusammenhangskomponenten:

Definition:

Für einen gegebenen gerichteten Graphen $G = (V, E)$ mit den Knotenmengen $V_1, \ldots, V_k$ für die k starken Zusammenhangskomponenten von G heißt der gerichtete Graph $G' = (V', E')$ mit $V' = \{u_1, \ldots, u_k\}$ und $E' = \{(u_i, u_j) \mid \exists v \in V_i, w \in V_j, (v, w) \in E\}$, der verdichteter Graph von G.

1. Berechne die Knotenmengen $V_1, \ldots, V_k$ der starken Zusammenhangskomponenten von G.
2. Berechne den verdichteten Graphen $G' = (V', E')$.
3. Berechne den transitiven Abschluss von G'.
4. Berechne den transitiven Abschluss $G^* = (V, E^*)$ von G mit Hilfe von G'. Füge alle Kanten $(u, v),\ u, v \in V_i,\ 1 \leq i \leq k$, und alle Kanten $(u, v),\ u \in V_i,\ v_{\in} V_j$, mit $(u_i, v_j) \in E'$, zu G hinzu.

Aufgabe 1 und 2 benötigen jeweils $O(|V| + |E|)$ viele Schritte. Aufgabe 3 kann in $O(k^3)$ vielen Schritten berechnet werden. Aufgabe 4 benötigt offenbar $O(|E^*|)$ viele Schritte.

Somit kann der transitive Abschluss für gerichtete Graphen in Zeit $O(|V|+|E^*|+k^3)$ berechnet werden.

Ein alternativer Algorithmus für die Berechnung der starken Zusammenhangs-Zusammenhangskomponenten

1. Bestimme mit einer Tiefensuche für jeden Knoten eine DFS-End-Nummer;

2. Betrachte alle Knoten $v_1, \ldots, v_n$ in absteigender Reihenfolge bzgl. ihrer DFS-End-Nummern.
3. Drehe alle Kanten in Ihrer Richtung um, d.h., ersetze jede Kante $(u, v) \in E$ durch die Kante (v, u).
4. Markiere alle Knoten als „unbesucht".
5. Starte eine Tiefensuche für $u = v_1, \ldots, v_n$ falls u noch nicht besucht wurde, ausschließlich über noch unbesuchte Knoten.
6. Markiere bei der Suche von u alle neu erreichbaren Knoten mit „besucht" und gebe sie als eine starke Zusammenhangskomponente aus.

Warum funktioniert dieser Algotrithmus?

Beweisidee: Sei MAX-DFE(u) die größte DFS-End-Nummer der Knoten in der starken Zusammenhangskomponente von u.

Wenn es in G einen Weg von einem Knoten u zu einem Knoten v gibt und $v$nicht in der gleichen starken Zusammenhangskomponente wie u ist, dann ist MAX-DFE(u) > MAX-DFE(v).

Sei $v_{\max}$ nun der Knoten mit größter DFS-End-Nummer, dann gibt es keinen Knoten u mit MAX-DFE(u) > MAX-DFE($v_{\max}$).

Gibt es nun einen Weg von u nach $v_{\max}$, dann muss u in der starken Zusammenhangskomponente von $v_{\max}$ sein, ansonsten wäre ja MAX-DFE(u) > MAX-DFE($v_{\max}$).

Gibt es andererseits keinen Weg von u nach $v_{\max}$, so kann u nicht in der starken Zusammenhangskomponente von $v_{\max}$ sein.

Ein Knoten u ist also genau dann in der starken Zusammenhangskomponente von $v_{\max}$, wenn es einen Weg von u nach $v_{\max}$ gibt.

6 Vorrangswarteschlangen (Priority-Queues)

Eine Priority-Queue D soll Elemente mit Schlüsseln speichern, auf denen über ihre Schlüssel eine Ordnung definiert ist. Folgende Operationen sollen zur Verfügung stehen:

- Initialisierung einer Datenstruktur D: $D = \text{init}(\ldots)$

  bzw. Initialisierung einer Datenstruktur D mit einem Element k: $D = \text{init}(k, \ldots)$
- Einfügen eines Elementes k in Datenstruktur D: $\text{insert}(D, k)$
- Entfernen eines Elementes k: $\text{delete}(D, k)$
- Element mit kleinstem Schlüssel in D bestimmen: $\text{access-min}(D)$
- Element mit kleinstem Schlüssel entfernen: $\text{delete-min}(D)$
- Verändern des Schlüssels x von Element k zu Schlüssel y: $\text{relocate}(D, k, y)$
- Erniedrigen des Schlüssels x von Element k auf Schlüssel y: $\text{decrease}(D, k, y)$
- Zusammenfügen von zwei Datenstrukturen D_1, D_2 zu einer neuen Datenstruktur D:
  $D = \text{merge}(D_1, D_2)$

Die Operation $\text{access-min}(D)$ soll in $O(1)$ Schritten, alle übrigen in $O(\log(N))$ Schritten ausführbar sein.

Bemerkungen

Die in der Datenstruktur gespeicherten Elemente speichern Informationen darüber, wo (und somit auch ob) sie sich in der Datenstruktur befinden. Es ist nicht notwendig, die Elemente in der Datenstruktur zu suchen!

Priority-Queues können auf verschiedene Arten implementiert werden, zum Beispiel durch Linksbäume, Binomial-Quees oder Fibonacci-Heaps.

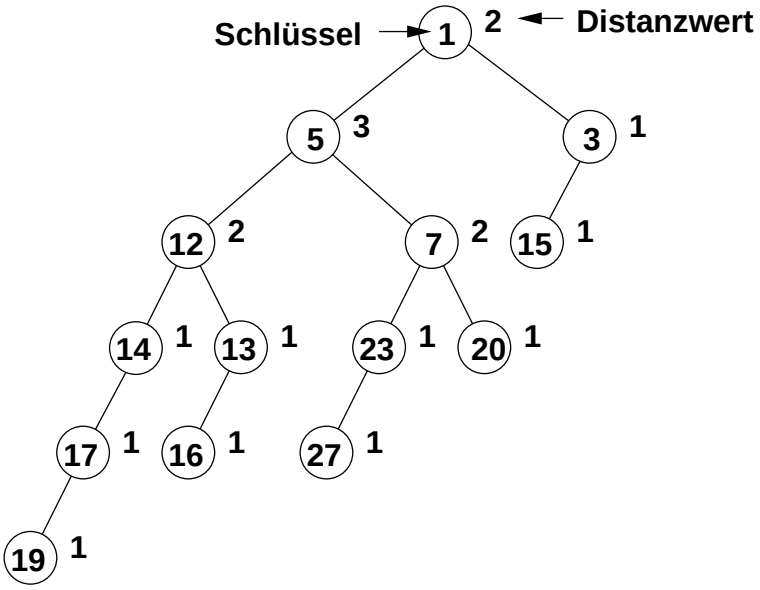
6.1 Linksbäume

Ein Baum mit $N + 1$ Blättern und N inneren Knoten ist balanciert, wenn jedes Blatt eine Tiefe aus $O(\log(N))$ hat.

Zur Implementierung von Priority-Queues reicht eine wesentlich schwächere Forderung aus, um zu sichern, dass die obigen Operationen in der vorgegebenen Zeit ausführbar sind.

Linksbäume sind binäre heapgeordnete, links-rechts geordnete Bäume, die in ihren inneren Knoten Elemente mit Schlüsseln speichern.

In den Zeichnungen lassen wir ab jetzt die Blätter weg.



1. Die Schlüsselwerte der Söhne sind stets größer als der Schlüsselwert des Vaters (Heapeigenschaft).
2. Jeder Knoten besitzt einen Distanzwert.
   (a) Der Distanzwert der Blätter ist 0.
   (b) Der Distanzwert an einem inneren Knoten ist der Distanzwert des rechten Sohnes plus 1.
   (c) Der Distanzwert des rechten Sohnes ist kleiner gleich als der Distanzwert des linken Sohnes.

Bemerkung:

In einem Linksbaum hat das rechteste Blatt eine Tiefe von $O(\log(N))$.

Alle Operationen lassen sich auf das Verschmelzen von zwei Linksbäumen zurückführen.

Die Operation insert(D, k)

Verschmelze D mit einem Linksbaum, der einen einzigen inneren Knoten k mit Distanz 1 hat.

Die Operation delete-min(D)

Entferne die Wurzel und verschmelze die beiden Teilbäume der Wurzel.

Die Operation delete(D, k)

Ersetze den Knoten k durch ein Blatt, das gegebenenfalls mit seinem Bruder vertauscht wird, um links den Teilbaum mit größerer Distanz anzuordnen. Adjustiere die Distanzwerte vom eingefügten Blatt bis zur Wurzel. Verschmelze den entstandenen Linksbaum und die beiden Teilbäume von k miteinander.

Die Operation relocate(D, k, y)

delete(D, k), ändere den Schlüssel von k auf y und führe anschließend insert(D, k) aus.

Die Operation decrease(D, k, y)

relocate(D, k, y)

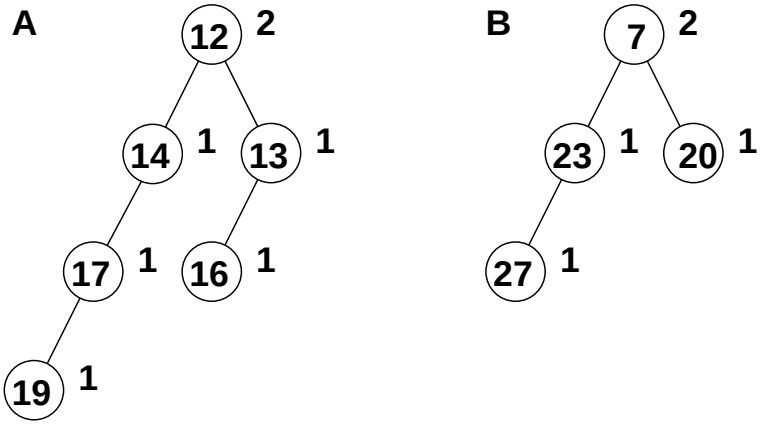
Die Operation merge(A, B)

Wenn A oder B keine Elemente speichert, also aus genau einem Blatt besteht, dann ist das Ergebnis der Linksbaum B bzw. A.

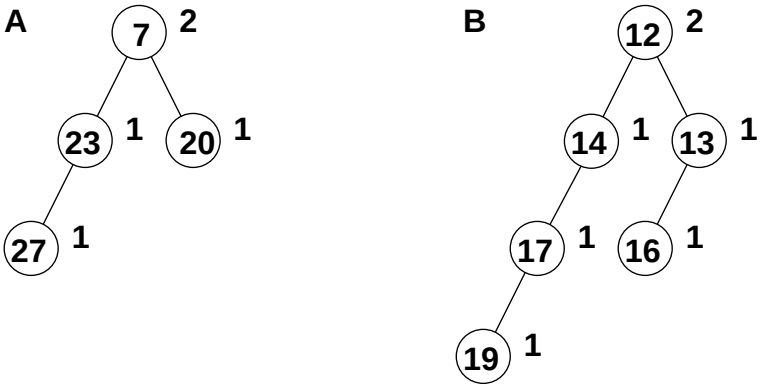
Ohne Beschränkung der Allgemeinheit sei der Schlüssel an der Wurzel von A kleiner als der Schlüssel an der Wurzel von B, ansonsten werden die beiden Linksbäume vertauscht. Zuerst wird rekursiv mit der gleichen Methode der rechte Teilbaum von A mit dem Linksbaum B vereinigt. Das Ergebnis ist der neue rechte Teilbaum von A. Ist die Distanz vom rechten Teilbaum von A größer als die Distanz vom linken Teilbaum von A, so werden die beiden Teilbäume von A vertauscht.

Beispiel:

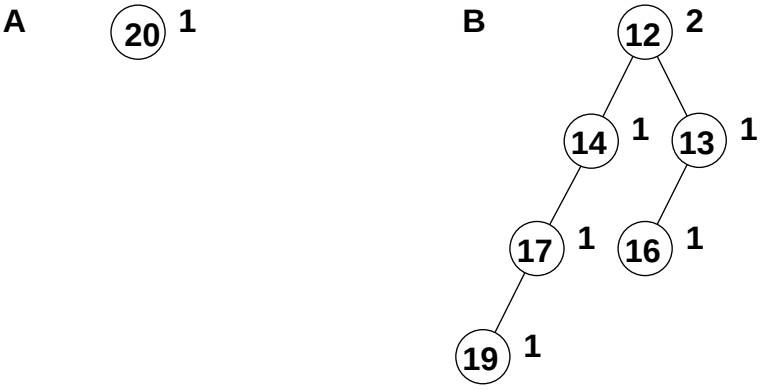
Verschmelzen zweier Linksbäume A und B mit merge(A, B):



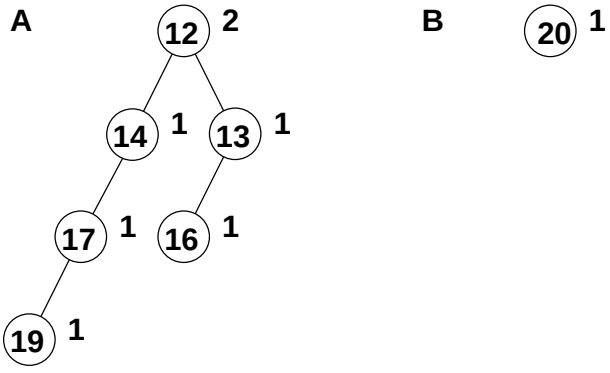
Bäume vertauschen:



Verschmelzen zweier Linksbäume A und B mit merge(A.rechts, B):

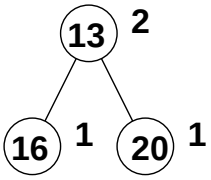


Bäume vertauschen:

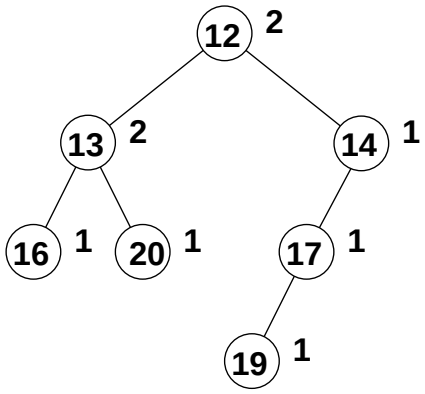


Verschmelzen zweier Linksbäume A und B mit merge(A.rechts, B):

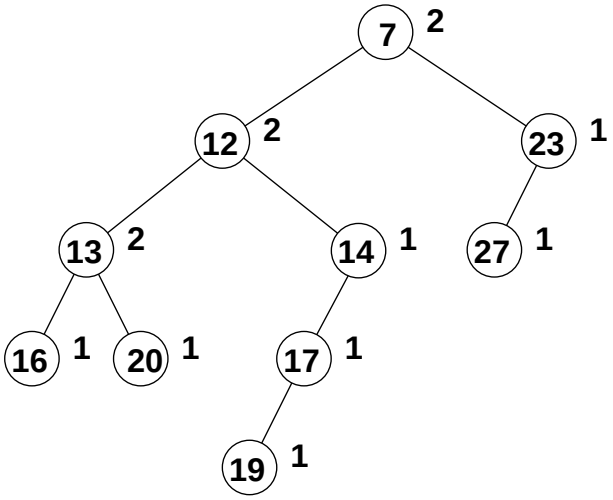
Ergebnis der letzten Rekursion:



Ergebnis der vorletzten Rekursion:



Ergebnis:



Die Laufzeit der Vereinigungsoperation ist beschränkt durch die Summe der Längen der beiden Pfade von der Wurzel zum jeweils rechtesten Blatt, also logarithmisch in der Anzahl der gespeicherten Elemente beschränkt.

Laufzeiten:

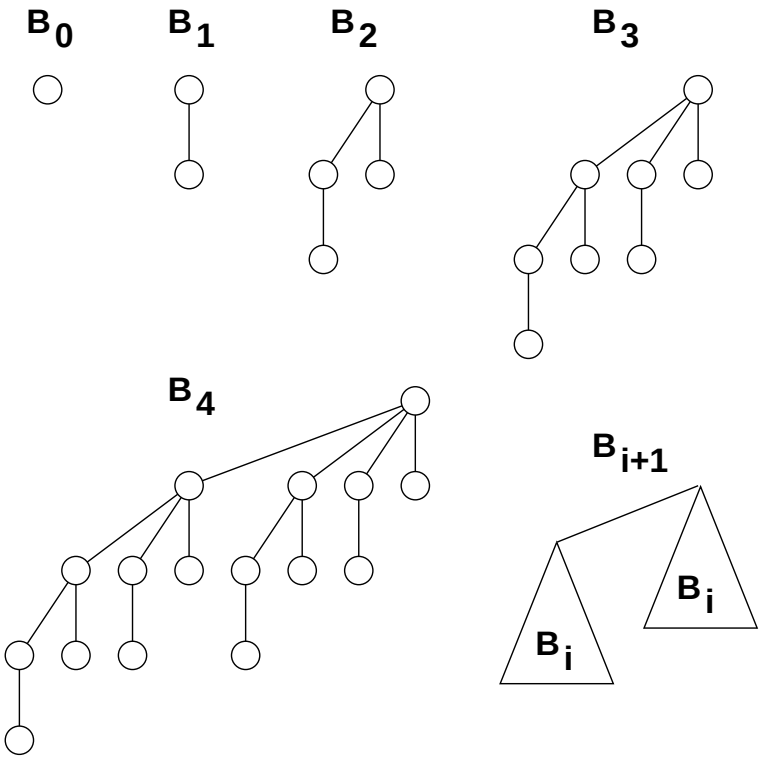
init()	$O(1)$
insert(D, k)	$O(\log(N))$
access-min(D)	$O(1)$
delete-min(D)	$O(\log(N))$
delete(D, k)	$O(\log(N))$
relocate(D, k, y)	$O(\log(N))$
decrease(D, k, y)	$O(\log(N))$
merge(D_1, D_2)	$O(\log(N))$

6.2 Binomial-Queues

Binomialbäume sind heapgeordnete Bäume, die in allen Knoten Elemente mit Schlüsseln speichern.

Ein Baum vom Typ B_0 besteht aus genau einem Knoten.

Ein Baum vom Typ B_{i+1} für $i \geq 0$ besteht aus zwei Kopien von Bäumen vom Typ B_i, indem man die Wurzel der einen Kopie zum Sohn der Wurzel der anderen macht.



Eigenschaften:

Ein Baum vom Typ B_i hat 2^i Knoten, die Höhe i und $\binom{i}{h-1}$ Knoten auf Höhe h.

Die i Teilbäume der Wurzel sind vom Typ $B_{i-1}, B_{i-2}, \ldots, B_0$.

Zur Speicherung einer Menge von N Elementen werden genau so viele Binomialbäume benötigt wie Einsen in der Binärdarstellung von $N = (d_{m-1} \ldots d_0)$ enthalten sind. Es wird genau dann ein Baum vom Typ B_j benötigt, wenn $d_j = 1$.

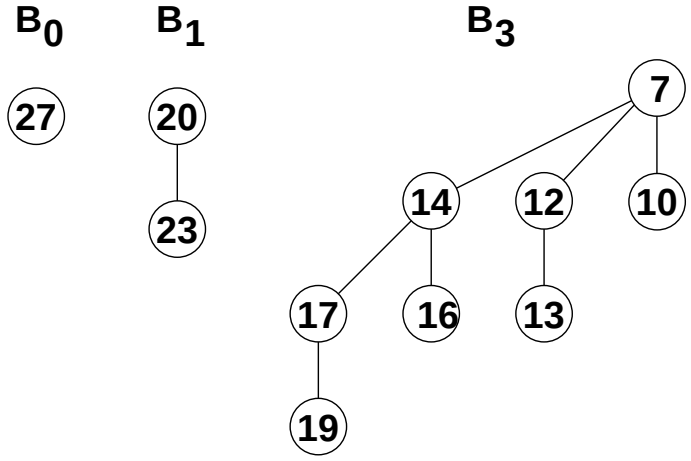
$\Longrightarrow$ Die Anzahl der Bäume in einer Binomial-Queue mit $N > 1$ Elementen ist höchstes $\log(N)$.

Beispiel: 11 (dezimal) $=$ $\underbrace{1}_{B_3}\ 0\ \underbrace{1}_{B_1}\ \underbrace{1}_{B_0}$ (binär)

Eine derartige Repräsentation einer Menge mit N Elementen ist eine Binomial-Queue vom Typ D_N.

Beispiel:

$N = 11$, $\{7, 10, 12, 13, 14, 16, 17, 19, 20, 23, 27\}$



Die Operation init()

Erzeuge einen Baum mit genau einen Knoten der k speichert.

Zeitaufwand $O(1)$

Die Operation access-min(D_N)

Durchsuchen der Wurzeln der Binomial-Bäume $\Rightarrow$ Zeitaufwand $O(\log(N))$

Die Operation merge(D_N, D_M)

(Das Vereinigen zweier Bäume vom gleichen Typ ist bereits definiert.)

Vereinige die Bäume aus beiden Binomial-Queues vom Typ D_N und D_M. Gibt es in der Vereinigung zwei Bäume vom Typ B_0, so vereinige sie zu einem Baum vom Typ B_1. Gibt es anschließend zwei Bäume vom Typ B_1, so vereinige sie zu einem Baum vom Typ B_2, usw.

(Addition zweier Zahlen nach der Schulmethode!)

$\Rightarrow$ Laufzeit: $O(\log(N + M))$

Die Operation insert(D_N, k)

merge(D_N, init(k))

Die Operation delete-min(D_N)

Sei B_j der Baum mit access-min(D_N) in der Wurzel.
Sei D_{N-2^j} der Wald D_N ohne B_j.
Sei D_{2^j-1} der Wald, der entsteht, wenn in B_j die Wurzel entfernt wird.

Verschmelze D_{N-2^j} mit D_{2^j-1} zu delete-min(D_N).

$\Rightarrow$ Laufzeit: $O(\log(N))$

Die Operation delete(D_N, k)

Sei B_j der Baum, der k enthält. Entferne B_j aus D_N. Sei D_{N-2^j} das Ergebnis. Zerlege nun B_j in einen Wald F.

Sei B^l_{j-1} der linke Teilbaum und B^r_{j-1} der rechte Teilbaum, aus dem B_j zusammengesetzt wurde.

Ist k in B^l_{j-1}, dann wird B^r_{j-1} in F aufgenommen und mit B^l_{j-1} fortgefahren.

Ist k in B^r_{j-1}, dann wird B^l_{j-1} in F aufgenommen und mit B^r_{j-1} fortgefahren.

Ist der Teilbaum mit Wurzel k erreicht, dann entferne k und füge die entstehenden Teilbäume zu F hinzu.

Vereinige D_{N-2^j} und F zu einer Binomial-Queue.

$\Rightarrow$ Laufzeit: $O(\log(N))$

Die Operation relocate(H, k, y)

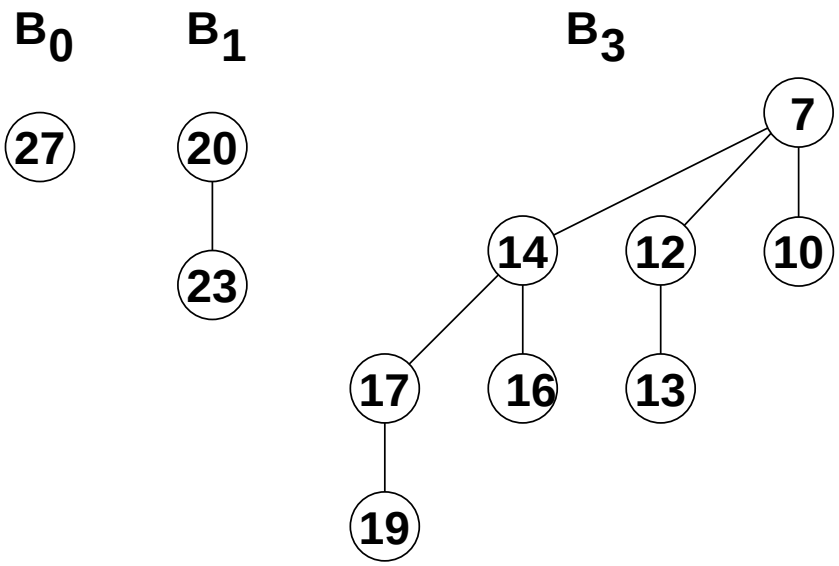
delete(D, k), ändere den Schlüssel von k auf y und führe anschließend insert(D, k) aus.

Bemerkung:

Die Implementation von Binomial-Queues verlangt die programmtechnische Realisation von Bäumen mit unbeschränktem Grad.

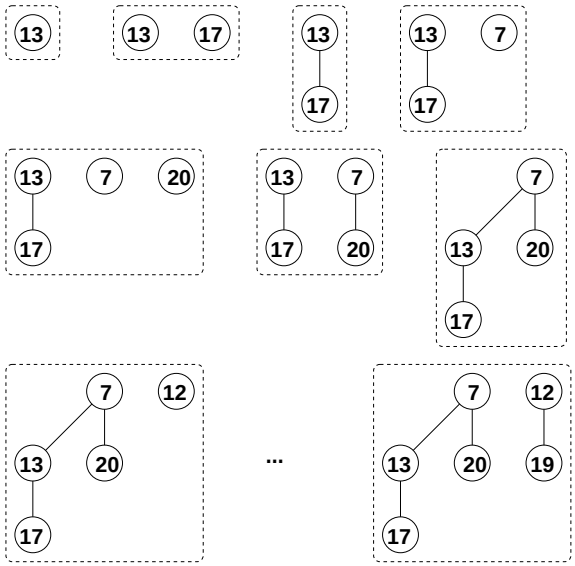
Beispiel:

Nacheinander werden folgende Elemente eingefügt: 13, 17, 7, 20, 12, 19



Beispiel:

Herausnahme von 13



Laufzeiten:

init(k)	$O(1)$
insert(D, k)	$O(\log(N))$
access-min(D)	$O(\log(N))$
delete-min(D)	$O(\log(N))$
delete(D, k)	$O(\log(N))$
relocate(D, k, y)	$O(\log(N))$
decrease(D, k, y)	$O(\log(N))$
merge(D_1, D_2)	$O(\log(N))$

6.3 Fibonacci-Heaps

Ein Fibonacci-Heap ist eine Sammlung heapgeordneter Bäume.

Die Struktur der Fibonacci-Heaps ist implizit durch die erklärten Operationen definiert, d.h., jede mit den bereitgestellten Operationen aufbaubare Struktur ist ein Fibonacci-Heap.

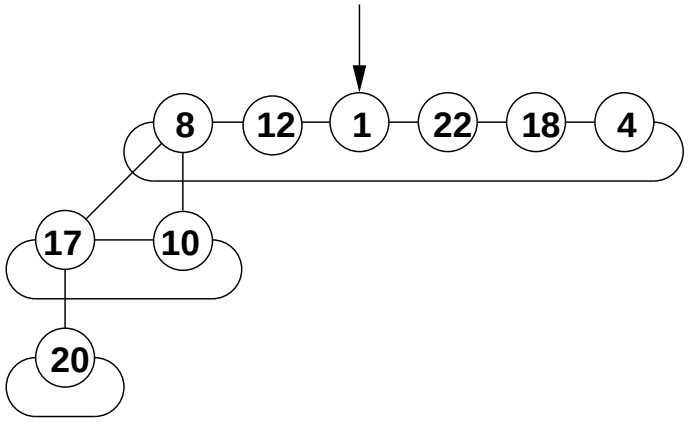
Die Wurzeln der Bäume sind Elemente einer doppelt verketteten zyklisch geschlossenen Wurzelliste.

Der Fibonacci-Heap besitzt einen Zeiger (Minimalzeiger) auf das kleinste Element (das Minimalelement) in der Wurzelliste.

Die Söhne jedes Knotens sind ebenfalls doppelt zyklisch verkettet.

Jeder Knoten hat einen Rang und ein Markierungsfeld. Der Rang entspricht der Anzahl der Söhne.

Beispiel:



Die Operation init()

Erzeuge einen leeren Fibonacci-Heap mit einem "Null"-Zeiger.

Die Operation access-min(H)

Der Minimalzeiger von H zeigt auf das Minimalelement, welches somit direkt angegeben werden kann.

Die Operation merge(H_1, H_2)

Hänge die Wurzellisten von H_1 und H_2 aneinander. Der neue Minimalzeiger wird auf das Minimum der Minimalelemente von H_1 und H_2 gesetzt.

Die Operation insert(H, k)

Erzeuge einen Fibonacci-Heap H', der nur das Element k enthält. Der Rang von k ist 0. Das Element k ist unmarkiert. Anschließend werden H und H' mit der Operation merge(H, H') vereinigt.

Bemerkung:

Alle bisherigen Operationen sind in Zeit $O(1)$ ausführbar!

Die Operation delete-min(H)

Entferne den Minimalknoten u aus der Wurzelliste und bilde eine neue Wurzelliste durch Einhängen der Liste der Söhne von u an Stelle von u.

Durchführbar in $O(1)$ Schritten.

Verschmelze nun solange zwei heapgeordnete Bäume, deren Wurzeln denselben Rang haben, zu einem neuen heapgeordneten Baum, bis die Wurzelliste nur Bäume enthält, deren Wurzeln einen verschiedenen Rang haben.

Beim Verschmelzen zweier Bäume B und B' vom Rang i entsteht ein Baum vom Rang $i + 1$. Ist das Element in der Wurzel v von B größer als das Element in der Wurzel v' von B', dann wird v ein Sohn von v' und das Markierungsfeld von v auf „unmarkiert“ gesetzt.

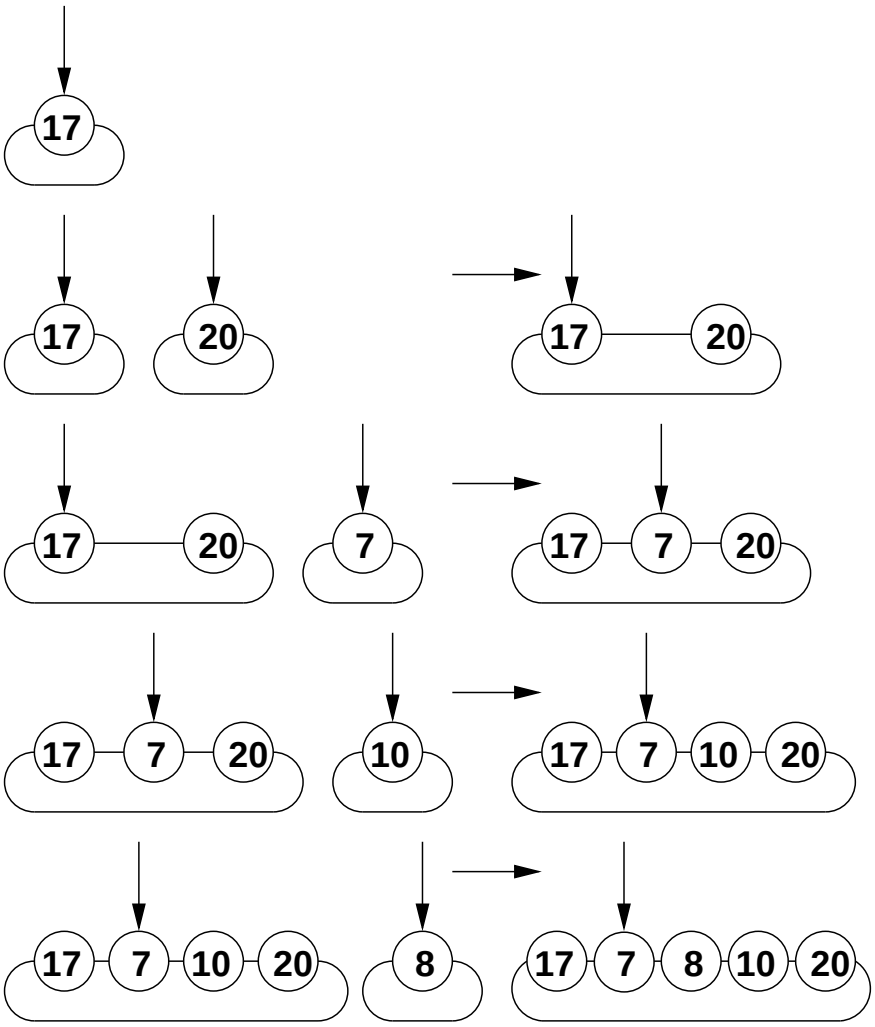
Aktualisiere dabei auch den Minimalzeiger.

(Dies entspricht der Schulmethode zur Addition von Binärzahlen.)

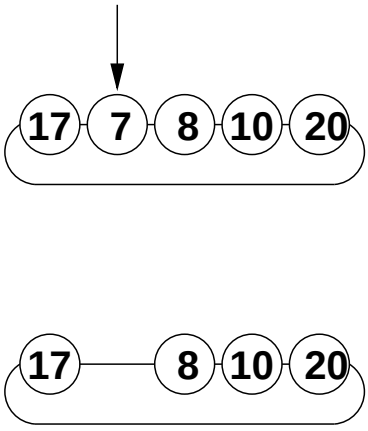
$\Rightarrow$ Worst-Case Laufzeit: $O(N)$

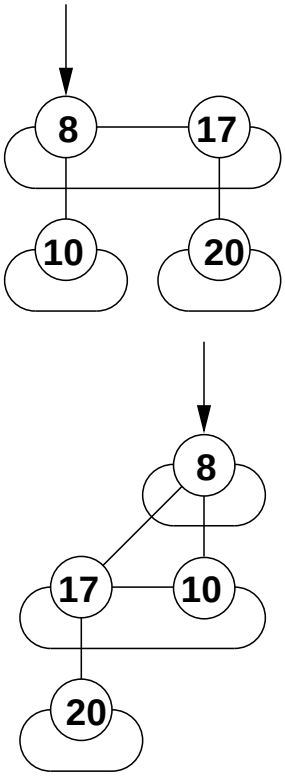
Beispiel:

Aufbau eines neuen Fibonacci-Heaps mit den Schlüsseln 17, 20, 7, 10, 8

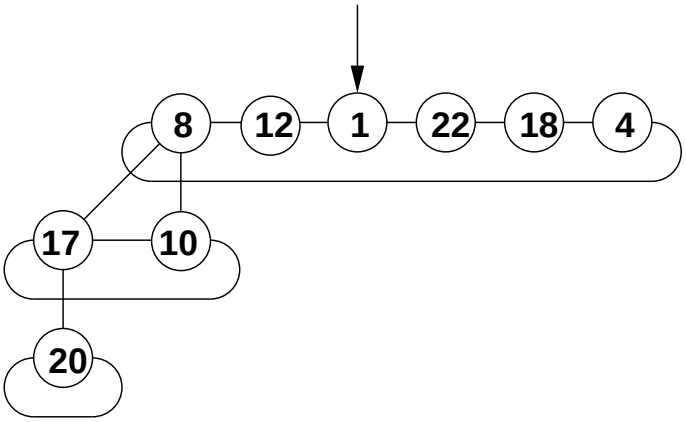

17
17
20
17
20
17
20
7
17
7
20
17
7
20
10
17
7
10
20
17
7
10
20
8
17
7
8
10
20

delete-min(H)

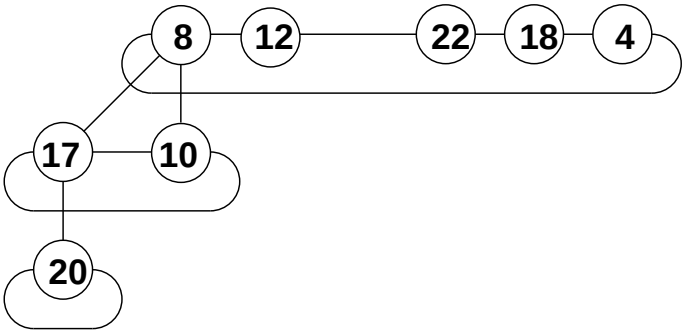

17
7
8
10
20
17
8
10
20

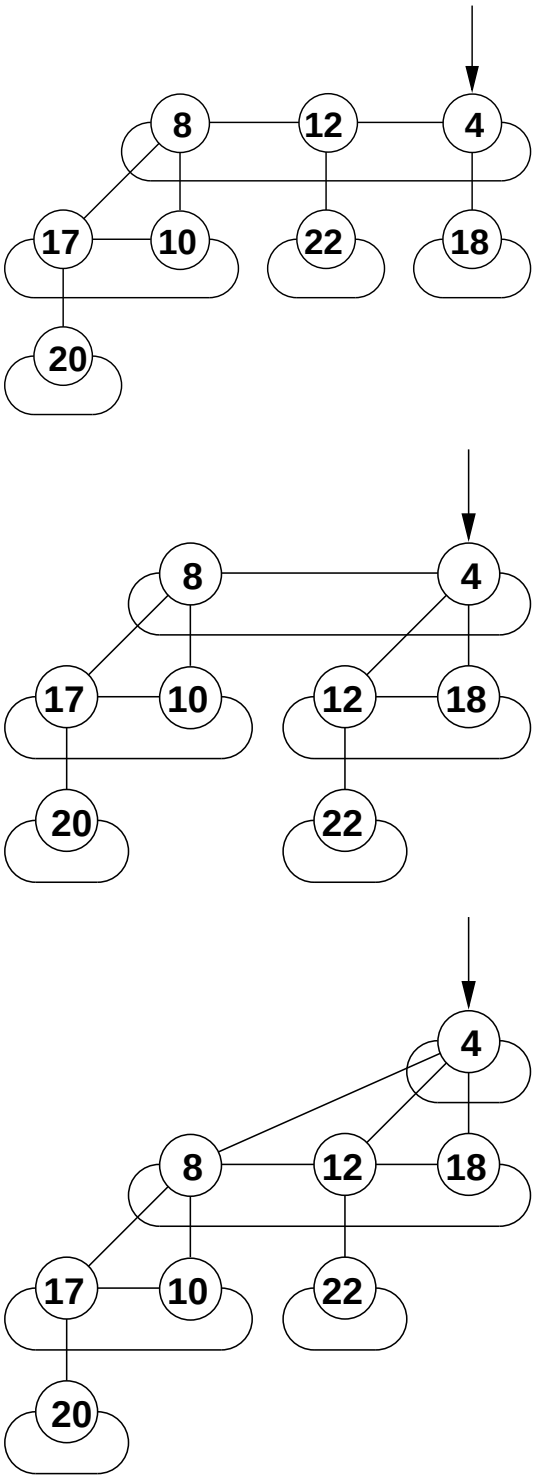


Einfügen der Schlüssel 12, 1, 22, 18 ,4 in den bestehenden Fibonacci-Heap



delete-min(H)





Beobachtungen:

Bis jetzt gilt:

1. Nach jeder $insert()$, $delete-min()$ und $merge()$-Operation sind die Bäume in der Wurzelliste Binomialbäume.
2. Nach jeder $delete-min()$-Operation bilden die Bäume in der Wurzelliste eine Binomial-Queue.

Die Operation decrease(H, k, y)

Trenne k von seinem Vater, erniedrige den Schlüssel auf y und hänge den heapgeordneten Baum in die Wurzelliste. Aktualisiere den Minimalzeiger.

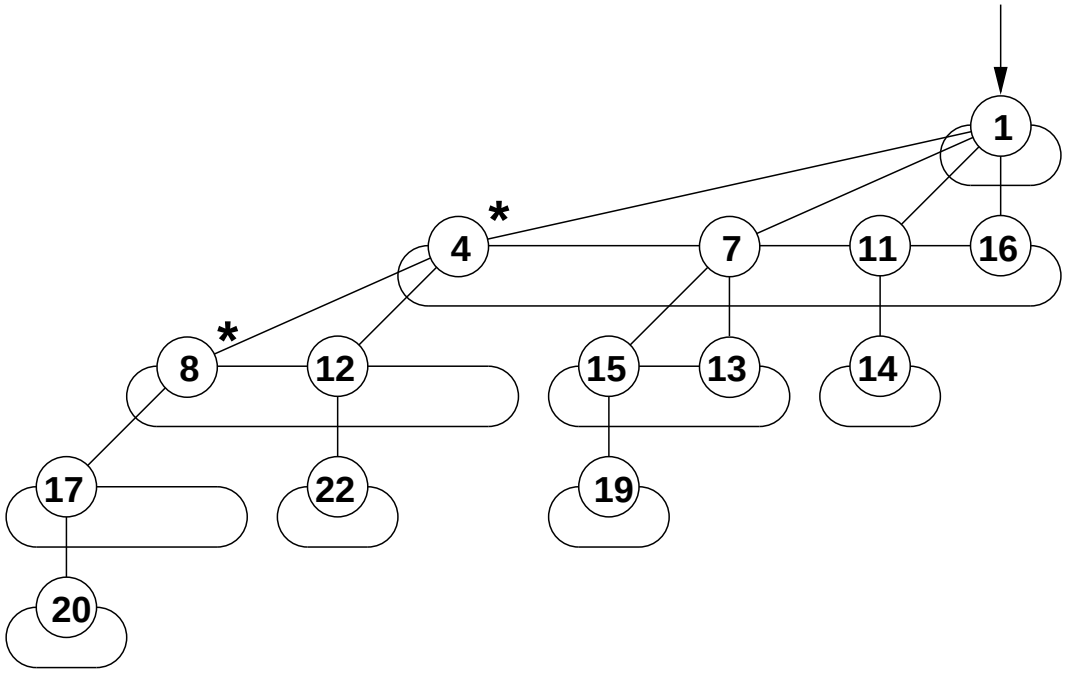
Durchführbar in $O(1)$ Schritten.

Es soll verhindert werden, dass mehr als zwei Söhne von einem Vater abgetrennt werden. Beim Abtrennen eines Knotens p von seinem Vater v wird v markiert. War v bereits markiert, wird auch v von seinem Vater v' abgetrennt und v' markiert, usw.

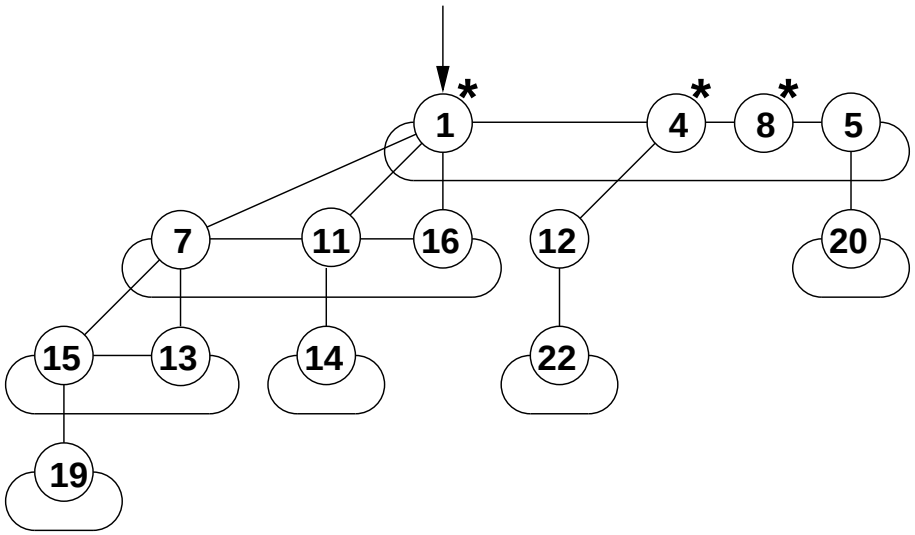
Alle abgetrennten Teilbäume werden in die Wurzelliste aufgenommen.

$\Rightarrow$ Worst-Case Laufzeit: $O(N)$ (geht nicht besser, Übungsaufgaben!)

Beispiel:



decrease$(H, 17, 5)$



Die Operation delete(H, k)

Setze k auf einen sehr kleinen Schlüssel und führe delete-min(H) aus.

$\Rightarrow$ Worst-Case Laufzeit: $O(N)$

Die Operation relocate(H, k, y)

delete(H, k), ändere den Schlüssel von k auf y und führe anschließend insert(D, k) aus.

Lemma:

Sei p ein Knoten eines Fibonacci-Heaps H.

Ordnet man die Söhne von p in der zeitlichen Reihenfolge, in der sie an p angehängt wurden, so gilt:

Der i-te Sohn von p hat mindestens den Rang $\max\{i-2, 0\}$.

Beweis:

Als der i-te Sohn u von p (zeitlich betrachtet) an p angehängt wurde, hatten u und p den gleichen Rang.
Dieser Rang war $i-1$, falls Knoten p bisher keinen Sohn verloren hat, der vor dem $i$ten -Sohn (zeitlich betrachtet) angehängt wurde, bzw. i, falls p bereits einen Sohn verloren hat, der zeitlich betrachtet vor dem i-ten Sohn angehängt wurde, bzw. $i-1+j$, mit $j \geq 2$ falls p bereits j Söhne verloren hat, die zeitlich betrachtet vor dem i-ten Sohn angehängt wurden (kann nur vorkommen, wenn p in der Wurzelliste ist). Also hat u mindestens den Rang $i-1$, falls u bisher noch keinen Sohn verloren hat. Da Knoten u höchstens einen Sohn verloren haben kann, da er sonst von p abgetrennt worden wäre, hat u mindestens den Rang $i-2$.

Lemma: Jeder Knoten p vom Rang k eines Fibonacci-Heaps H ist Wurzel eines Teilbaums mit mindestens F_{k+2} Knoten, wobei F_{k+2} die $(k+2)$-te Fibonacci-Zahl ist.

Beweis: (induktiv)

Für $k \geq 2$ gilt

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &= F_k + F_{k-1} + F_k \\ &= F_{k-1} + F_{k-2} + F_{k-1} + F_k \\ &= F_{k-2} + F_{k-3} + F_{k-2} + F_{k-1} + F_k \\ &= F_2 + F_1 + F_2 + \cdots F_{k-1} + F_k \\ &= 2 + \sum_{i=2}^{k} F_i \end{aligned}$$

Sei S_k die minimale Anzahl der Knoten in einem Teilbaum mit Wurzel p vom Rang k in einem Fibonacci-Heap.

$S_0 \geq 1 = F_2$

$S_1 \geq 2 = F_3$

Sei p ein Knoten vom Rang $k \geq 2$. Ordne die k Söhne in der Reihenfolge, in der sie an p angehängt wurden.

Der erste Sohn von p hat mindestens den Rang 0.

Nach obigem Lemma hat der i-te Sohn mindestens den Rang $\max\{i-2, 0\}$.

Mit p zusammen folgt aus einer einfachen Induktion für $i \geq 2$:

$S_i \geq$		1		Knoten p
	$+$	1		Knoten im ersten Teilbaum mit Rang ≥ 0
	$+$	1	$(\geq S_0 \geq F_2)$	Knoten im 2-ten Teilbaum mit Rang ≥ 0
	$+$	2	$(\geq S_1 \geq F_3)$	Knoten im 3-ten Teilbaum mit Rang ≥ 1
	$+$	3	$(\geq S_2 \geq F_4)$	Knoten im 4-ten Teilbaum mit Rang ≥ 2
	$+$	5	$(\geq S_3 \geq F_5)$	Knoten im 5-ten Teilbaum mit Rang ≥ 3
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$+$	x	$(\geq S_{i-2} \geq F_i)$	Knoten im i-ten Teilbaum mit Rang $\geq i-2$

und somit $S_k \geq 2 + \sum_{i=0}^{k-2} S_i \geq 2 + \sum_{i=2}^{k} F_i = F_{k+2}$. $\square$

Bemerkung:

Da die Fibonacci-Zahlen exponentiell mit einer Basis von etwa 1.618 wachsen, hat jede Wurzel in einem Fibonacci-Heap mit N Knoten einen Rang $k \in O(\log(N))$.

Daraus folgt, dass die Anzahl der Bäume in der Wurzelliste nach einer delete-min()-Operation aus $O(\log(N))$ ist.

Worst-Case Laufzeiten:

init(k)	$O(1)$
insert(H, k)	$O(1)$
access-min(H)	$O(1)$
delete-min(H)	$O(N)$
delete(H, k)	$O(N)$
relocate(H, k, y)	$O(N)$
decrease(H, k, y)	$O(N)$
merge(H_1, H_2)	$O(1)$

Die Operationen delete-min(), delete(), relocate(), decrease() in Fibonacci-Heaps haben zwar eine schlechte Worst-Case-Laufzeit, aber dafür eine sehr gute amortisierte Laufzeit (siehe nächstes Kapitel).

Satz:

Für das Ausführen von N Operationen beginnend mit einem leeren Fibonacci-Heap H ist die insgesamt benötigte tatsächliche Zeit beschränkt durch die gesamte amortisierte Zeit, wobei

1. die amortisierten Zeiten der delete-min(H)-, delete(H, k)-, und relocate(H, k, y)-Operation aus $O(\log(N))$ sind und
2. die amortisierten Zeiten aller anderen Operationen aus $O(1)$ sind.

Das sollten Sie sich merken!

7 Amortisierte Laufzeitanalysen

Bestimme die maximalen durchschnittlichen (amortisierten) Kosten pro Operation für eine beliebige Folge von Operationen!

Das Bankkonto-Paradigma:

Betrachte eine Folge von m Operationen.

Wir ordnen jedem Bearbeitungszustand einen nichtnegativen Kontostand und jeder Operation amortisierte Kosten zu.

Sei Φ_i der Kontostand nach der Ausführung der i-ten Operation.

Dann sind die amortisierten Kosten a_i der i-ten Operation die tatsächlichen Kosten t_i plus die Differenz der Kontostände:

$$a_i \quad = \quad t_i + \Phi_i - \Phi_{i-1}$$

Die gesamten amortisierten Kosten sind:

$$\begin{array}{rcl} \sum_{i=1}^{m} a_i & = & a_1 + a_2 + a_3 + \cdots + a_m \\ & = & \quad t_1 + \Phi_1 - \Phi_0 \\ & & + \quad t_2 + \Phi_2 - \Phi_1 \\ & & + \quad t_3 + \Phi_3 - \Phi_2 \\ & & + \quad \vdots \\ & & + \quad t_m + \Phi_m - \Phi_{m-1} \\ & = & \quad \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0 \end{array}$$

also

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0$$

Φ_0 ist der Anfangskontostand und Φ_m der Endkontostand.

Ist $\Phi_0 \leq \Phi_m$, so ist der gesamte zur Ausführung der m Operationen benötigte amortisierte Aufwand $\sum_{i=1}^{m} a_i$ eine obere Schranke für den tatsächlichen Aufwand.

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - (\Phi_m - \Phi_0) \leq \sum_{i=1}^{m} a_i$$

7.1 Selbstanordnende Listen

Wir betrachten eine Liste mit N Elementen, auf der wir eine Folge $s = s_1, \ldots, s_m$ von m Zugriffsoperationen ausführen.

Sei A ein beliebiger Algorithmus zur Selbstanordnung der Elemente, der das Element zuerst sucht und es dann durch Vertauschungen einige Positionen zum Anfang oder zum Ende der Liste bewegt.

Der Zugriff auf das Element auf Position k soll genau k Einheiten kosten. Eine Vertauschung in Richtung Listenanfang soll kostenfrei sein. Eine Vertauschung in Richtung Listenende soll genau eine Einheit kosten.

Sei $C_A(s_i)$ die Schrittzahl zur Ausführung der Zugriffsoperation s_i, $1 \leq i \leq m$, und $C_A(s)$ die gesamte Schrittzahl zur Ausführung aller Zugriffsoperationen der Folge s.

Sei $F_A(s_i)$ bzw. $F_A(s)$ die Anzahl der kostenfreien Vertauschungen und $X_A(s_i)$ bzw. $X_A(s)$ die Anzahl kostenbehafteten Vertauschungen.

Unsere Algorithmen MF (Move-to-Front), T (Transpose) und FC (Frequently-Count) führen keine kostenbehafteten Vertauschungen durch. Somit gilt

$$X_{\mathrm{MF}}(s) = X_{\mathrm{T}}(s) = X_{\mathrm{FC}}(s) = 0$$

Das Element auf Position k kann anschließend maximal mit allen $(k-1)$ vorangehenden Elementen kostenfrei vertauschen werden. Für jede Operation ist also die Anzahl der kostenfreien Vertauschungen höchstens so groß wie die Kosten der Operation minus 1. Daher gilt für jede Strategie A (mit m Operationen)

$$\begin{aligned} F_A(s) \;=\; F_A(s_1) + \cdots F_A(s_m) \;&\leq\; C_A(s_1) - 1 \\ &+\; C_A(s_2) - 1 \\ &\;\vdots \\ &+\; C_A(s_m) - 1 \\ &=\; C_A(s) - m \end{aligned}$$

also

$$F_A(s) \leq C_A(s) - m.$$

Satz: Für jeden Algorithmus A der obigen Art und für jede Folge s von m Zugriffsoperationen gilt:

$$C_{\mathrm{MF}}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Dieser Satz besagt grob (falls die kostenbehafteten Vertauschungen nicht größer sind als die kostenfreien Vertauschungen $+m$), dass die MF-Regel höchstens doppelt so schlecht ist, wie jede andere Regel zur Selbstanordnung von Listen.

Beweis:

Für zwei Listen L_1, L_2, die die selben Elemente in möglicherweise unterschiedlicher Anordnung enthalten, sei $\text{inv}(L_1, L_2)$ die Anzahl der Inversionen von Elementen in L_2 bzgl. L_1, d.h., $\text{inv}(L_1, L_2)$ ist die Anzahl der Elementpaare x_i, x_j die deren Anordnung in L_2 eine andere ist als in L_1

Beispiel:

$$\begin{aligned} L_1 &: 4, 3, 5, 1, 7, 2, 6 \\ L_2 &: 3, 6, 2, 5, 1, 4, 7 \end{aligned}$$

$$\text{inv}(L_1, L_2) = 12$$

In L_2 steht 3 vor 4, 6 vor 2, 6 vor 5, 6 vor 1, 6 vor 4, 6 vor 7, 2 vor 5, 2 vor 1, 2 vor 4, 2 vor 7, 5 vor 4 und 1 vor 4. Diese Elementpaare stehen jedoch in L_1 in umgekehrter Reihenfolge.

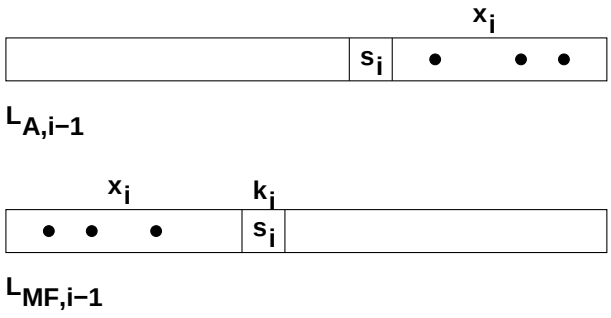
Sei $L_{A,i}$ die Liste, die Algorithmus A nach dem i-ten Zugriff hinterlässt und $L_{\text{MF},i}$ die Liste, die die MF-Regel nach dem i-ten Zugriff hinterlässt.

Der Kontostand nach dem i-ten Zugriff ist definiert durch $\text{inv}(L_{A,i}, L_{\text{MF},i})$.

Der initiale Kontostand ist $\text{inv}(L_{A,0}, L_{\text{MF},0}) = 0$, da beide Algorithmen mit der selben Liste beginnen.

Der Kontostand ist immer nicht negativ.

Betrachte nun den Zugriff auf das Element s_i (den i-ten Zugriff). Sei k_i die Position, an der das Element s_i in $L_{\text{MF},i-1}$ steht. Sei x_i die Anzahl der Elemente, die in $L_{\text{MF},i-1}$ vor s_i aber in $L_{A,i-1}$ hinter s_i stehen.



Jedes dieser x_i Elemente ist mit s_i eine Inversion.

Nun gilt

$$\text{inv}(L_{A,i-1}, L_{\text{MF},i}) \quad = \quad \text{inv}(L_{A,i-1}, L_{\text{MF},i-1}) \quad - x_i \quad + (k_i - 1 - x_i),$$

da durch das Vorziehen von s_i in $L_{\text{MF},i-1}$ insgesamt x_i Inversionen verschwinden und $k_i - (1 + x_i)$ Inversionen entstehen. $k_i - (1 + x_i)$ ist die Anzahl der übrigen Elemente in $L_{\text{MF},i-1}$) vor dem Element s_i.

Jedes anschließende Vorziehen von s_i in $L_{A,i-1}$ mit Algorithmus A erniedrigt die Anzahl der Inversionen.

Jedes anschließende Vertauschen von s_i in $L_{A,i-1}$ zum Listenende mit Algorithmus A erhöht die Anzahl der Inversionen.

Somit gilt

$$\text{inv}(L_{A,i}, L_{\text{MF},i}) \quad = \quad \text{inv}(L_{A,i-1}, L_{\text{MF},i-1}) \quad - x_i \quad + (k_i - 1 - x_i) - F_A(s_i) + X_A(s_i)$$

Da die tatsächlichen Kosten für den Zugriff auf s_i mit der MF-Regel k_i sind, erhält man für die amortisierten Kosten a_i der i-ten Operation

$$\begin{aligned} a_i &= k_i + \text{inv}(L_{A,i}, L_{\text{MF},i}) - \text{inv}(L_{A,i-1}, L_{\text{MF},i-1}) \\ &= k_i - x_i + (k_i - 1 - x_i) - F_A(s_i) + X_A(s_i) \\ &= 2(k_i - x_i) - 1 - F_A(s_i) + X_A(s_i) \end{aligned}$$

Da die tatsächlichen Kosten für den Zugriff auf s_i in $L_{A,i-1}$ mit Algorithmus A mindestens $k_i - x_i$ sind ($k_i - x_i - 1$ Elemente stehen in beiden Listen vor dem Element s_i, also auch in der Liste $L_{A,i-1}$ vor s_i), folgt

$$\sum_{i=1}^{m} a_i \leq 2C_A(s) + X_A(s) - F_A(s) - m$$

Da die Summe der amortisierten Kosten eine obere Schranke für die Summe der tatsächlichen Kosten sind, folgt die Aussage des Satzes.

7.2 Splay-Bäume

Bei Splay-Bäumen werden alle drei Operationen (Suchen, Einfügen, Entfernen) auf die Splay-Operation zurückgeführt.

Wir messen die Kosten einer Splay-Operation durch die Anzahl der ausgeführten Rotationen (plus 1, falls keine Operation ausgeführt wird).

Jede zig-Operation zählt eine Rotation und jede zig-zig- und zig-zag-Operation zählt zwei Rotationen.

Wir ordnen jedem Splay-Baum, der durch eine Folge von Operationen erzeugt wurde, einen Kontostand zu.

Manchmal werden viele, ein anderes Mal werden wenige Rotationen ausgeführt.

Führen wir eine billige Operation (mit wenigen Rotationen) durch, so sparen wir etwas, was wir dem Konto gutschreiben.

Führen wir eine teure Operation (mit vielen Rotationen) durch, so können wir einen erforderlichen Mehraufwand dem Konto entnehmen, falls das Konto ein Guthaben enthält.

Für einen Knoten p, sei $s(p)$ die Anzahl aller inneren Knoten (Schlüssel) im Teilbaum mit Wurzel p.

Sei $r(p)$ der Rang von p, definiert durch $r(p) = \log(s(p))$.

Für einen Baum T mit Wurzel p und für einen in p gespeicherten Schlüssel x sei $r(T)$ und $r(x)$ definiert als $r(p)$.

Der Kontostand $\Phi(T)$ eines Suchbaumes T sei die Summe aller Ränge der inneren Knoten von T.

Lemma: Der amortisierte Aufwand der Operation splay(T, x) ist höchstens $3 \cdot (r(T) - r(x)) + 1$.

Beweis: Ist x in der Wurzel von T gespeichert, so wird nur auf x zugegriffen und keine weitere Operation ausgeführt. Der tatsächliche Aufwand ($= 1$) stimmt mit dem amortisierten Aufwand überein und das Lemma gilt.

Angenommen es wird wenigstens eine Rotation durchgeführt.

Für jede bei der Ausführung von splay(T, x) durchgeführte Rotation, die einen Knoten p betrifft, betrachten wir die Größe $s(p)$ und den Rang $r(p)$ von p unmittelbar vor und die Größe $s'(p)$ und den Rang $r'(p)$ von p unmittelbar nach Ausführung der Rotation.

Wir werden zeigen, dass jede zig-zig- und zig-zag-Operation auf p mit amortisiertem Aufwand $3(r'(p) - r(p))$ und jede zig-Operation mit amortisiertem Aufwand $3(r'(p) - r(p)) + 1$ ausführbar ist.

Angenommen obige Aussage wäre bereits gezeigt.

Sei $r^{(i)}(x)$ der Rang von x nach Ausführung der i-ten von insgesamt k zig-zig-, zig-zag- oder zig-Operationen. (Nur die letzte Operation kann eine zig-Operation sein.)

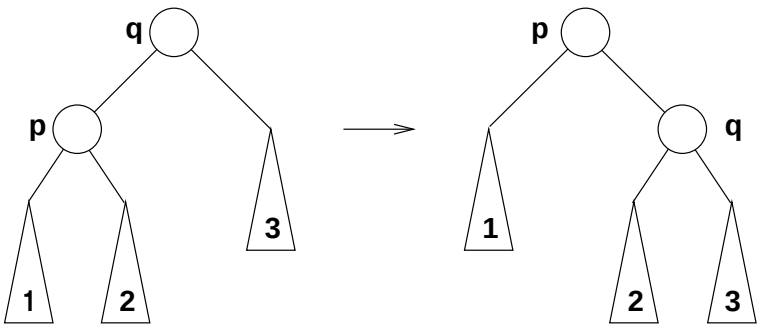
Dann ergibt sich als amortisierter Aufwand von splay(T, x)

$$\begin{aligned} & 3(r^{(1)}(x) - r(x)) \\ + \; & 3(r^{(2)}(x) - r^{(1)}(x)) \\ \vdots \; & \\ + \; & 3(r^{(k)}(x) - r^{(k-1)}(x)) + 1 \\ = \; & 3(r^{(k)}(x) - r(x)) + 1 \end{aligned}$$

Da x durch die k Operationen zur Wurzel gewandert ist, gilt $r^{(k)}(x) = r(T)$ und damit das Lemma.

Beweis von: Jede zig-zig- und zig-zag-Operation auf p ist mit amortisiertem Aufwand $\leq 3(r'(p) - r(p))$ und jede zig-Operation mit amortisiertem Aufwand $\leq 3(r'(p) - r(p)) + 1$ ausführbar.

Fall 1: (zig-Operation)



Dann ist $q = \varphi(p)$ die Wurzel. Es wird eine Rotation ausgeführt. Die tatsächlichen Kosten sind 1. Es können höchstens die Ränge von p und q geändert worden sein.

Die amortisierten Kosten der zig-Operation sind daher:

$$\begin{aligned}
a_{zig} &= 1 + \Phi'(T) - \Phi(T) \\
&= 1 + (r'(p) + r'(q) + \cdots) - (r(p) + r(q) + \cdots) \\
&= 1 + (r'(p) + r'(q)) - (r(p) + r(q)) \\
&= 1 + r'(q) - r(p), \text{ weil } r'(p) = r(q) \\
&\leq 1 + r'(p) - r(p), \text{ weil } r'(p) \geq r'(q) \\
&\leq 1 + 3(r'(p) - r(p)), \text{ weil } r'(p) \geq r(p)
\end{aligned}$$

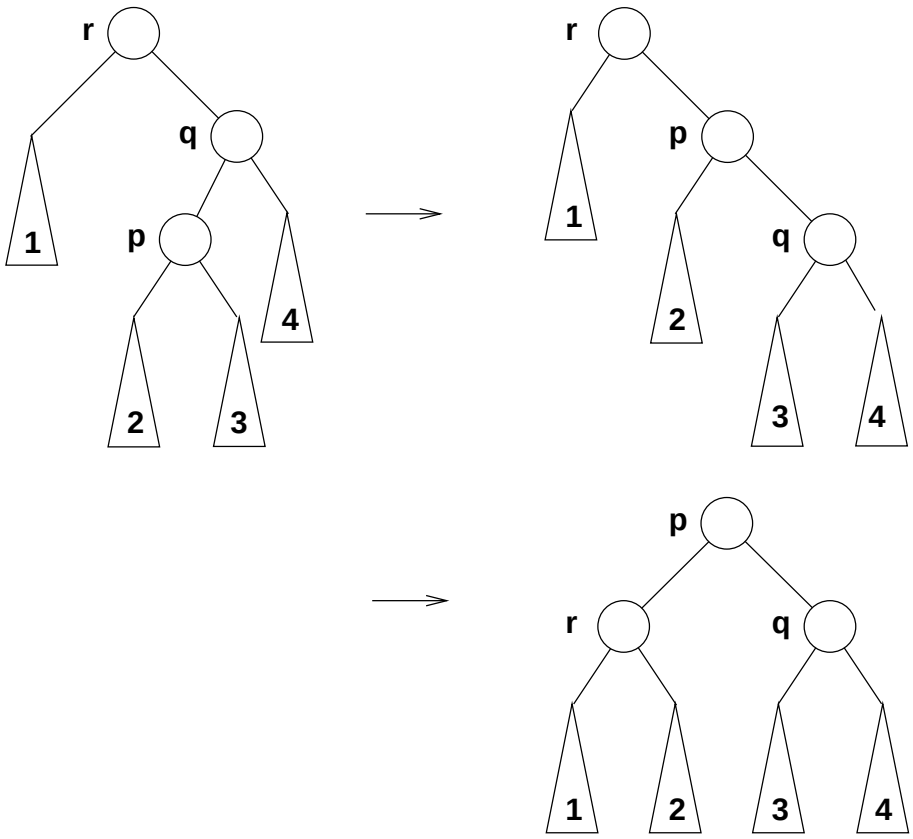
Für die nächsten beiden Fälle, formulieren wir folgende Hilfsaussage:

Sind a und b positive Zahlen und gilt $a+b \leq c$, so folgt $\log(a)+\log(b) \leq 2\log(c)-2$.

(Das geometrische Mittel zweier positiver Zahlen ist niemals größer als das arithmetische Mittel.)

$$\begin{aligned}
\sqrt{ab} &\leq (a+b)/2 \\
\sqrt{ab} &\leq c/2 \\
ab &\leq (c/2)(c/2) \\
\log(ab) &\leq \log((c/2)(c/2)) \\
\log(a) + \log(b) &\leq 2\log(c/2) \\
\log(a) + \log(b) &\leq 2\log(c) - 2\log(2) \\
\log(a) + \log(b) &\leq 2\log(c) - 2
\end{aligned}$$

Fall 2: (zig-zag-Operation)



Sei $q = \varphi(p)$ und $r = \varphi(\varphi(p))$.

Die Operation hat tatsächliche Kosten von 2, weil zwei Rotationen durchgeführt werden.

Es können sich höchstens die Ränge von p, q und r ändern, ferner ist $r'(p) = r(r)$.

Also gilt für die amortisierten Kosten:

$$\begin{aligned} a_{zig-zag} &= 2 + \Phi'(T) - \Phi(T) \\ &= 2 + (r'(p) + r'(q) + r'(r) + \cdots) - (r(p) + r(q) + r(r) \cdots) \\ &= 2 + (r'(p) + r'(q) + r'(r)) - (r(p) + r(q) + r(r)) \\ &= 2 + r'(q) + r'(r) - r(p) - r(q) \end{aligned}$$

Weil p vor Ausführung der zig-zag-Operation Sohn von q war, gilt $r(q) \geq r(p)$ und somit

$$a_{zig-zag} \leq 2 + r'(q) + r'(r) - 2r(p)$$

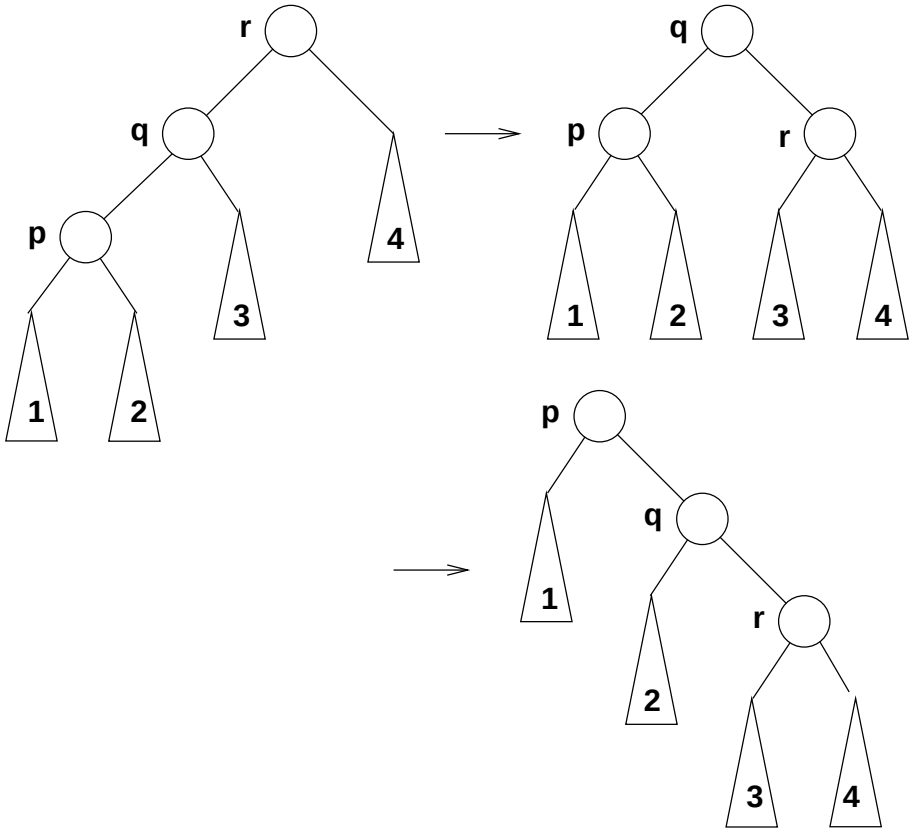
Ferner gilt $s'(q) + s'(r) \leq s'(p)$. Aus der Definition des Ranges und obiger Hilfsaussage folgt

$$r'(q) + r'(r) \leq 2r'(p) - 2,$$

und somit

$$\begin{aligned} a_{zig-zag} &\leq 2(r'(p) - r(p)) \\ &\leq 3(r'(p) - r(p)), \text{ weil } r'(p) \geq r(p) \end{aligned}$$

Fall 3: (zig-zig-Operation)



Sei wieder $q = \varphi(p)$ und $r = \varphi(\varphi(p))$.

Die Operation hat tatsächliche Kosten von 2, weil zwei Rotationen durchgeführt werden.

Also gilt wie im vorherigen Fall $r'(p) = r(r)$ und für die amortisierten Kosten:

$$\begin{aligned} a_{zig-zig} &= 2 + \Phi'(T) - \Phi(T) \\ &= 2 + (r'(p) + r'(q) + r'(r) + \cdots) - (r(p) + r(q) + r(r) \cdots) \\ &= 2 + (r'(p) + r'(q) + r'(r)) - (r(p) + r(q) + r(r)) \\ &= 2 + r'(q) + r'(r) - r(p) - r(q) \end{aligned}$$

Da vor Ausführung der zig-zig-Operation p Sohn von q und nachher q Sohn von p ist, folgt $r(p) \leq r(q)$ und $r'(p) \geq r'(q)$, und somit

$$a_{zig-zig} \leq 2 + r'(p) + r'(r) - 2r(p)$$

Diese Summe ist genau dann kleiner oder gleich $3(r'(p)-r(p))$, wenn $r(p)+r'(r) \leq 2r'(p)-2$.

Aus der zig-zig-Operation folgt, dass $s(p)+s'(r) \leq s'(p)$. Mit obiger Hilfsaussage und der Definition der Ränge erhält man die obige Ungleichung.

Damit ist das Lemma bewiesen.

Bemerkung:

Nur im Fall 3 (der zig-zig-Operation) ist die Abschätzung der amortisierten Kosten scharf.

Folgerung:

Satz: Das Ausführen einer beliebigen Folge von m Such-, Einfüge, Entferne-Operationen, in der höchstens N mal Einfügen vorkommt und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $O(m \cdot log(N))$ Schritte.

Weil für jeden im Verlauf der Operationsfolge erzeugten Baum $s(T) \leq N$ gilt und jede Operation ein konstantes Vielfaches der Kosten der Splay-Operation verursacht, folgt die Behauptung aus obigem Lemma.

7.3 Fibonacci-Heaps

Definition der Kontostandsfunktion $\Phi()$:

Der Kontostand $\Phi(H)$ für einen Fibonacci-Heap H sei die Anzahl der Bäume in der Wurzelliste plus zwei mal die Anzahl der markierten Knoten, die nicht in der Wurzelliste auftreten.

(Für mehrere Fibonacci-Heaps sei der Gesamtkontostand die Summe der Kontostände der einzelnen Fibonacci-Heaps.)

$\Phi(H) = 0$, falls H noch keine Elemente speichert.

$\Phi(H)$ ist niemals negativ.

Die Operation insert(H, k):

Die tatsächlichen Kosten sind aus $O(1)$, der Kontostand erhöht sich um 1, die amortisierten Kosten sind aus $O(1)$.

Die Operation access-min(H):

Die tatsächlichen Kosten sind aus $O(1)$, der Kontostand bleibt unverändert, die amortisierten Kosten sind aus $O(1)$.

Die Operation merge(H_1, H_2):

Die tatsächlichen Kosten sind aus $O(1)$, der Gesamtkontostand bleibt unverändert, die amortisierten Kosten sind aus $O(1)$.

Die Operation delete-min(H)

Die Laufzeit von delete-min(H) ist von der Anzahl der Verschmelze-Operationen und somit von der Anzahl der Knoten in der Wurzelliste abhängig.

Das Verschmelzen zweier Bäume vom gleichen Rang kostet eine Einheit, welche durch das Verschwinden eines Baumes aus der Wurzelliste ausgeglichen wird.

Die Anzahl der markierten Knoten, die nicht in der Wurzelliste auftreten, bleibt entweder unverändert oder nimmt ab (falls markierte Knoten in die Wurzelliste aufgenommen werden).

Sei $w(H)$ die Anzahl der Bäume in der Wurzelliste von H vor der Ausführung von delete-min(H). Sei $x(H)$ die Anzahl der markierten Knoten in H, die nicht in der Wurzelliste auftreten. Sei rang(k) der Rang des Minimalelementes k.

Die Wurzelliste wird um maximal rang(k) $\in O(\log(N))$ Knoten erweitert und einmal durchlaufen.

Die tatsächlichen Kosten der delete-min(H) Operation betragen somit rang(k) + $w(H)$.

Sei H' der entstehende Heap. Dann gilt $w(H') \in O(\log(N))$ und $x(H') \leq x(H)$.

Also ändert sich der Kontostand von $w(H) + 2x(H)$ auf $w(H') + 2x(H')$.

$\Rightarrow$ amortisierte Gesamtlaufzeit:

$$\text{rang}(k) + w(H) + \ (w(H') + 2x(H')) \ - \ (w(H) + 2x(H)) \ \in \ O(\log(N))$$

Die Operation decrease(H, k, y):

Die Laufzeit von decrease(H, k, y) ist von der Anzahl der indirekten Abtrennungen (der markierten Knoten) abhängig.

Jede Abtrennung eines Knotens verursacht eine Kosteneinheit.

1. Wird ein Knoten von seinem unmarkierten Vater abgetrennt, dann kostet dies eine Einheit. Durch die Aufnahme in die Wurzelliste und die Markierung des Vaters nimmt der Kontostand um 3 Einheiten zu.

   Dies kommt höchstens ein Mal vor!

   Tatsächliche Kosten plus Kontostandsdifferenz = 4.

2. Wird ein Knoten von seinem markierten Vater abgetrennt, dann kostet dies ebenfalls eine Einheit. Die Anzahl der Knoten in der Wurzelliste erhöht sich um 1. Die Anzahl der markierten Knoten erniedrigt sich um 1.

   Tatsächliche Kosten plus Kontostandsdifferenz = 0.

Insgesamt: amortisierte Gesamtlaufzeit: $O(1)$

Amortisierte Laufzeitabschätzung für delete(H, k)

delete(H, k) = decrease(H, k, y) und delete-min(H)

$\Rightarrow$ amortisierte Gesamtlaufzeit: $O(\log(N))$

Satz: Führt man, beginnend mit dem anfangs leeren Fibonacci-Heap H, eine beliebige Folge s von m Operationen aus, dann ist die dafür insgesamt benötigte Zeit beschränkt durch die gesamte amortisierte Zeit, wobei

1. die amortisierte Zeit einer einzelnen delete-min(H)-Operation aus $O(\log(N))$ ist,
2. die amortisierte Zeit einer einzelnen delete(H, k)-Operation aus $O(\log(N))$ ist und
3. die amortisierten Zeiten aller anderen Operationen aus $O(1)$ sind.

Das sollten Sie sich merken!

8 Graphalgorithmen

8.1 Kürzeste Wege

Notationen:

Wir betrachten gerichtete Graphen $G = (V, E)$ mit einer Kostenfunktion $f : E \to \mathbb{R}$, die jeder Kante einen reeller Kostenwert zuordnet. (Auch negative Kostenwerte sind erlaubt!)

Ein Graph $G = (V, E)$ mit einer reellwertigen Kostenfunktion $f : E \to \mathbb{R}$ heißt kantenbewerteter Graph, oder einfach bewerteter Graph.

Die Länge $L_f(p)$ eines Weges $p = (v_1, \ldots, v_k)$ in einem kantenbewerteten Graphen G mit Kantenbewertung f ist die Summe der Kostenwerte aller Kanten des Weges.

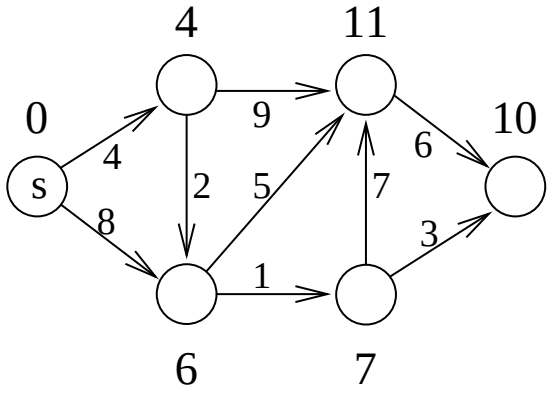
$$L_f(p) := \begin{cases} \sum_{i=1}^{k-1} f(\{u_i, u_{i+1}\}) & \text{falls } k \geq 2 \\ 0 & \text{sonst} \end{cases}$$

Die kürzeste Weglänge $\mathrm{kw}(u, v)$ von einem Knoten u zu einem Knoten v ist die minimale Länge aller Wege von u nach v, falls mindestens ein solcher Weg existiert, ansonsten ist die kürzeste Weglänge ∞.

$$\mathrm{kw}(u, v) = \min(\{L_f(p) \mid p \text{ ist ein Weg von } u \text{ nach } v\} \cup \{\infty\}).$$

Ein Weg p von u nach v mit $L_f(p) = \mathrm{kw}(u, v)$ ist ein kürzester Weg von u nach v.

Beispiel:



Problemvarianten:

1. Single-Pair

   Gegeben ist ein kantenbewerteter gerichteter Graph und zwei Knoten u, v.

   Gesucht ist ein kürzester Weg von u nach v.

2. Single-Source

   Gegeben ist ein kantenbewerteter gerichteter Graph und ein Knoten u.

   Gesucht ist für jeden Knoten v ein kürzester Weg von u nach v.

3. All-Pairs

   Gegeben ist ein kantenbewerteter gerichteter Graph.

   Gesucht ist für jedes Knotenpaar u, v ein kürzester Weg von u nach v.

Lemma:

1. Für jeden kürzesten Weg $p = (v_1, \ldots, v_k)$ von v_1 nach v_k ist jeder Teilweg $p' = (v_i, v_{i+1}, \ldots, v_j)$, $1 \leq i \leq j \leq k$, ein kürzester Weg von v_i nach v_j.

   Beweisidee: Gäbe es einen kürzeren Teilweg p'' von v_i nach v_j, dann könnte in p der Teilweg p' durch p'' ersetzt werden, und der entstehende Weg von v_1 nach v_k wäre kürzer als p (Widerspruch!).

2. Es gibt genau dann einen kürzesten Weg von u nach v, wenn es einen Weg von u nach v gibt und kein Weg von u nach v enthält einen Kreis negativer Länge.

   Beweisidee: $\Rightarrow$ Wenn es einen kürzesten Weg p von u nach v gibt, dann gibt es einen Weg von u nach v, nämlich p, und jeder Weg von u nach v enthält keinen Kreis negativer Länge, da es sonst einen Weg von u nach v gibt, der kürzer ist als p.

   Beweisidee: $\Leftarrow$ Angenommen jeder Weg p von u nach v enthält keinen Kreis negativer Länge. Dann gibt es für jeden Weg p von u nach v einen einfachen Weg p' von u nach v dessen Weglänge kleiner oder gleich der Weglänge von p ist. (Ein einfacher Weg besucht jeden Knoten höchstens ein mal.) Da die Anzahl der einfachen Wege zwischen zwei Knoten endlich ist, muss einer von ihnen ein kürzester Weg sein, wenn es mindestens einen Weg von u nach v gibt.

Daraus ergibt sich die folgende Invariante für alle Algorithmen, die zu bereits berechneten kürzesten Wegen Schritt für Schritt Kanten hinzufügen.

1. Für jedes Knotenpaar u, w und jede Kante $(v, w) \in E$ mit $\text{kw}(u, v) < \infty$ gilt:

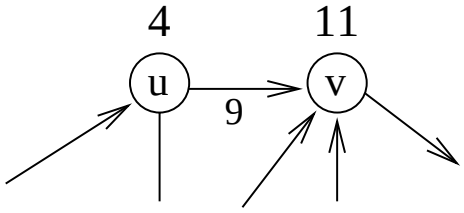
$$\text{kw}(u, v) + f((v, w)) \geq \text{kw}(u, w)$$

2. Für jedes Knotenpaar u, w mit $\text{kw}(u, w) < \infty$ gibt es eine Kante $(v, w) \in E$ mit:

$$\text{kw}(u, v) + f((v, w)) = \text{kw}(u, w)$$

Beispiel:

Hier ist die Dreiecksungleichung mit $4 + 9 \geq 11$ erfüllt.



Definition: Sei $G = (V, E)$ ein kantenbewerteter gerichteter Graph mit einer Kostenfunktion $f : E \to \mathbb{R}$. Eine Distanzfunktion für einen Knoten $s \in V$ ist eine Funktion

$$d_s : V \longrightarrow \mathbb{R} \cup \{\infty\}$$

mit

- $d_s(s) = 0$
- $\forall u \in V - \{s\}$ ist $d_s(u)$ der Kostenwert eines Weges von s nach u.
  Falls kein Weg von s nach u existiert, ist $d_s(u) = \infty$.

Eine Distanzfunktion $d_s : V \longrightarrow \mathbb{R} \cup \{\infty\}$ für s ist eine Kürzeste-Wege-Funktion für s, falls $d_s(u) = kw(s, u)$ für alle $u \in V$.

Satz: Sei $G = (V, E)$ ein gerichteter Graph mit Kostenfunktion $f : E \longrightarrow \mathbb{R}$ und sei $s \in V$. Sei $d_s : V \longrightarrow \mathbb{R} \cup \{\infty\}$ eine Distanzfunktion für s.

d_s ist genau dann eine Kürzeste-Wege-Funktion für s, wenn jede Kante $(u, v) \in E$ mit $d_s(v) < \infty$ die folgende Dreiecksungleichung erfüllt

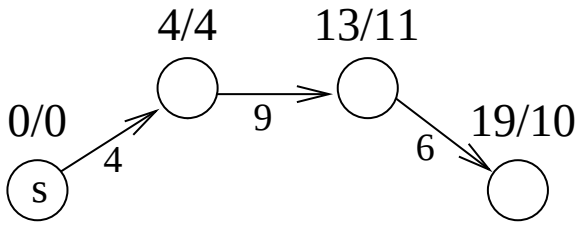
$$d_s(u) + f((u, v)) \geq d_s(v).$$

Beweis: Wir zeigen, d_s ist genau dann **keine** Kürzeste-Wege-Funktion für s, wenn es eine Kante (u, v) mit $d_s(v) < \infty$ gibt, so dass

$$d_s(u) + f((u, v)) < d_s(v).$$

$\Rightarrow$ Sei u ein Knoten mit $d_s(u) > \text{kw}(s,u)$. Der Fall $d_s(u) < \text{kw}(s,u)$ kann nicht vorkommen, da d_s eine Distanzfunktion ist. Betrachte nun einen kürzesten Weg von s nach u. Entlang des Weges muß es eine Kante (u',v') geben mit $kw(s,u) = d_s(u') + f((u',v')) < d_s(v')$, ansonsten wäre $d_s(u) = \text{kw}(s,u)$.
$\Leftarrow$ Wenn es eine Kante (u,v) mit $d_s(v) < \infty$ gibt, so dass $d_s(u) + f((u,v)) < d_s(v)$, dann ist $\text{kw}(s,v) < d_s(v)$, und somit d_s keine Kürzeste-Wege-Funktion für s. $\square$

Beispiel:



Wegealgorithmen konstruieren systematisch Distanzfunktionen, die alle Dreiecksungleichungen erfüllen.

Ein einfacher Kürzeste-Wege-Algorithmus:

Gegeben: Ein kantenbewerteter gerichteter Graph $G = (V,E)$ mit Kostenfunktion $f : E \to \mathbb{R}$ und ein Startknoten $s \in V$.

Gesucht: Eine Kürzeste-Wege-Funktion $d_s : V \to \mathbb{R} \cup \{\infty\}$ für s.

Berechne zuerst mit einer Tiefensuche von Knoten s eine Distanzfunktion d_s für den Startknoten s. Verändere die Distanzfunktion anschließend mit folgendem Algorithmus.

```
(1)  fertig := false;
(2)  while (!fertig) {
(3)       fertig := true;
(4)       for all (u,v) ∈ E {
(5)            if (d_s(u) + f((u,v)) < d_s(v)) {
(6)                 fertig := false;
(7)                 d_s(v) := d_s(u) + f((u,v)); } } }
```

Analyse:

1. Der obige Algorithmus terminiert nicht, wenn der Graph negative Kreise enthält, die vom Startknoten s erreichbar sind.

2. Es werden ausschließlich Distanzfunktionen berechnet. Wenn es einen Weg von s nach u der Länge $d_s(u)$ gibt und eine Kante $(u, v) \in E$ existiert, dann gibt es auch einen Weg von s nach v der Länge $d_s(u) + f((u, v))$.

3. Die schlussendlich berechnete Distanzfunktion ist eine Kürzeste-Wege-Funktion, da alle Dreiecksungleichungen erfüllt sind.

4. Die Laufzeit ist von der Reihenfolge abhängig, in der die Kanten betrachtet werden. Es ist leicht einzusehen, dass der obige Algorithmus spätestens nach $|V| \cdot |E|$ Überprüfungen der Dreiecksungleichung terminiert, falls keine negativen Kreise erreichbar sind. (Übungsaufgabe).

Algorithmisches Gerüst:

Wir merken uns für jeden Knoten v die bisher berechnete, vorläufige Entfernung $d[v]$ vom Startknoten s und den Vorgänger $p[v]$ von v, über den der Distanzwert $d[v]$ zuletzt aktualisiert wurde.

(1) for all $v \in V$ {
(2) $d[v] := \infty$;
(3) $p[v] :=$ undefined; }
(4) $d[s] := 0$;
(5) $p[s] := s$;
(6) while ($\exists (u, v) \in E$ mit $d[u] + f((u, v)) < d[v]$) {
(7) $d[v] := d[u] + f((u, v))$;
(8) $p[v] := u$; }

Wenn alle Dreiecksungleichungen erfüllt sind, ist

$$s, \ldots, p[p[v]], p[v], v$$

ein kürzester Weg von s nach v. Die Kanten $(p[v], v)$ bilden den sogenannten Kürzeste-Wege-Baum.

Beitrag von Ford (1956)

Satz: Sei $e_1, \ldots, e_k$ die Folge der ersten k getesteten Kanten.

Dann ist $d[u]$ mit $e_k = (w, u)$ der Kostenwert eines kürzesten Weges von s nach u, der eine Teilfolge von $e_1, \ldots, e_k$ ist. Eine Teilfolge von $e_1, \ldots, e_k$ ist jede Kantenfolge $e_{i_1}, \ldots, e_{i_l}$ mit $1 \leq i_1 < i_2 < \ldots < i_l \leq k$.

Beweisidee: Ist ein kürzesten Weg p von s nach u eine Teilfolge von $e_1, \ldots, e_k$, dann wurden die Distanzwerte der Knoten von p in der Reihenfolge des Weges p aktualisiert. $\square$

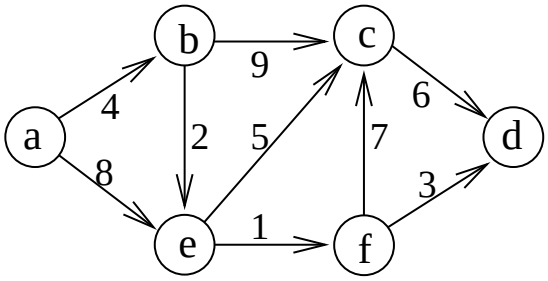
Aufgabe: Suche eine Kantenfolge, die alle einfachen Wege enthält.

Beispiel: Gerichtete, kreisfreie Graphen.

Betrachte eine topologische Sortiere die Kanten. (Kante (u, v) steht vor Kante (u', v')), wenn es in G einen Weg von v zu u' gibt.)

Nun ist jeder Weg in dem Graphen ein Teilweg der topologisch sortierten Kantenfolge.

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



Topologische Kantensortierung: $(a, b), (a, e), (b, c), (b, e), (e, c), (e, f), (f, c), (f, d), (c, d)$

Beitrag von Bellman (1959)

Verwalte die Kanten, die die Dreiecksungleichung verletzen könnten, in einer Datenstruktur D. Wir verwalten jedoch Knoten in D, wobei $v \in D$ bedeutet, dass alle Kanten (v, w) die Dreiecksungleichung verletzen könnten.

Bellman verwendete als Datenstruktur eine FIFO-Queue.

```
(1)   Bellman(s)
(2)   {
(3)       for all u ∈ V {
(4)           d[v] := ∞;
(5)           p[v] := undefined; }
(6)       d[s] := 0;
(7)       p[s] := s;
(8)       initialisiere eine FIFO-Queue D;
(9)       insert(D, s);
(10)      while (D ≠ ∅) {
(11)          entferne den ersten Knoten v aus D;
```

```
(12)            for all (v, w) ∈ E {
(13)                if (d[v] + f((v, w)) < d[w]) {
(14)                    d[w] := d[v] + f((v, w));
(15)                    p[w] := v;
(16)                    if (w ∉ D) { insert(D, w); } } } }
(17)  }
```

Analyse:

1. Der obige Algorithmus terminiert nicht, wenn der Graph negativen Kreise enthält, die vom Startknoten s erreichbar sind.
2. Der Algorithmus terminiert nach spätestens $|V| \cdot |E|$ Überprüfungen der Dreiecksungleichung, falls keine negativen Kreise erreichbar sind (Übungsaufgabe!).

Beitrag von Dijkstra (1959)

Idee: Verwalte die Randknoten in einer Prioritätswarteschlange

Die Datenstruktur D soll folgende Operationen unterstützen:

1. Einfügen eines Knotens v mit Schlüssel $d[v]$, insert(D, v).
2. Entfernen des Knotens v mit minimalem Schlüssel $d[v]$, extract-min(D).
3. Erniedrigen des Schlüssels $d[v]$ eines bereits aufgenommenen Knotens v, decrease$(D, v, d[v])$.

Eine solche Datenstruktur heißt Prioritätswarteschlange (priority queue).

```
(1)   Dijkstra(s)
(2)   {
(3)       for all u ∈ V {
(4)           d[v] := ∞;
(5)           p[v] := undefined; }
(6)       d[s] := 0;
(7)       p[s] := s;
(8)       initialisiere eine Datenstruktur D;
(9)       insert(D, s);
(10)      while (D ≠ ∅) {
```

```
(11)            v := extract-min(D);
(12)            for all (v, w) ∈ E {
(13)                if (d[v] + f((v, w)) < d[w]) {
(14)                    d[w] := d[v] + f((v, w));
(15)                    p[w] := v;
(16)                    if (w ∉ D) { insert(D, w); }
(17)                    else decrease(D, w, d[w]); } } }
(18)    }
```

Beobachtungen:

- Dijkstras Algorithmus terminiert nicht bei der Existens von negativen Kreisen, die von s erreichbar sind.
- Dijkstras Algorithmus terminiert nach spätestens $|V| \cdot |E|$ Überprüfungen der Dreiecksungleichung, falls keine negativen Kreise erreichbar sind (Übungsaufgabe!).
- Angenommen die Implementierung der Priority-Queue erfolgt mit Fibonacci-Heaps.

  Wenn der Graph keine negativen Kantengewichte hat, dann wird jeder Knoten genau einmal in D eingefügt (kostet insgesamt für alle Einfügungen $O(|V|)$ Schritte) und genau einmal aus D entfernt (kostet zusammen für alle Entfernungen $O(|V| \cdot \log(|V|))$ Schritte), und höchstens $|E|$ mal in D neu angeordnet (kostet zusammen für alle Neuanordnunge $O(|E|)$ Schritte).

  Daraus ergibt sich für Dijkstras Algorithmus eine worst-case Laufzeit von

  $$O(|E| + |V| \cdot \log(|V|)),$$

  falls der bewertete Graph keine negativen Kantengewichte hat. (Laufzeit für insert(D, k), im Worst-Case aus $O(1)$, für extract-min(D), amortisiert aus $O(\log(N))$, für decrease(D, k, y), amortisiert aus $O(1)$, siehe Kapitel 8 (Fibonacci-Heaps).)

Einen gerichteten bewerteten Graphen, der keine negativen Kantengewichte hat, nennt man einen Distanzgraphen.

8.2 All-Pairs Kürzeste Wege

Lösungsvorschläge:

1. Löse für jeden Knoten das Single-Source Kürzeste-Wege-Problem

   Laufzeit: $O(|V|^2 \cdot |E|)$ für allgemeine Graphen und $O(|V|^2 \cdot \log(|V|) + |V| \cdot |E|)$ für Distanzgraphen

2. Algorithmus von Floyd-Warshall

   Sei A die Distanzmatrix für $G = (V, E)$ mit Kostenfunktion $f : E \longrightarrow \mathbb{R}$ und $V = \{u_1, \ldots, u_n\}$.

$$a_{i,j} = \begin{cases} f((u_i, u_j)) & \text{falls } (u_i, u_j) \in E \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

   Wegealgorithmus:

```
for k := 1,...,n {
    for i := 1,...,n {
        for j := 1,...,n {
            a_{i,j} := min{a_{i,j}, a_{i,k} + a_{k,j}}; } } }
```

   Invariante:

   Wenn die beiden inneren For-Schleifen mit den Laufvariablen i und j durchlaufen sind, dann ist $a_{i,j}$ der Kostenwert eines kürzesten Weges von u_i nach u_j, der nur über Knoten $u_1, \ldots, u_k$ läuft. Dies ist leicht induktiv zu zeigen.

   Laufzeit: $O(|V|^3)$

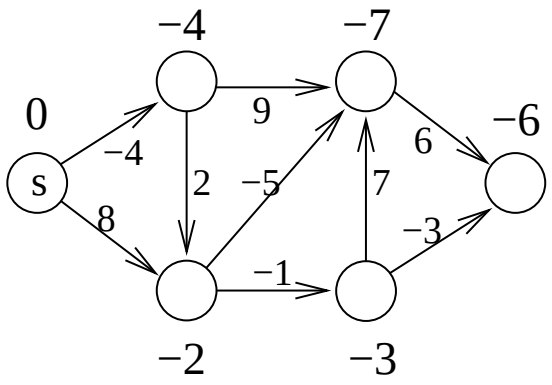
3. Transformiere G in einen Distanzgraphen.

   Sei $d_s : V \to \mathbb{R} \cup \infty$ eine Kürzeste-Wege-Funktion für s.

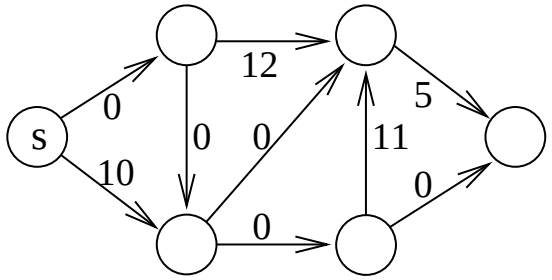
   Betrachte folgende Kostenfunktion $f' : E \longrightarrow \mathbb{R}$ für G:

$$f'((u, v)) = f((u, v)) - (d_s(v) - d_s(u)) = f((u, v)) + d_s(u) - d_s(v)$$

   Beispiel:



Der Distanzgraph sieht dann wie folgt aus:



Satz:

(a) Für alle Kanten $(u, v) \in E$ gilt: $f'((u, v)) \geq 0$

(b) Sei $d_u : V \longrightarrow \mathbb{R} \cup \infty$ eine Kürzeste-Wege-Funktion für u in G bzgl. Kostenfunktion f.
Sei $d'_u : V \longrightarrow \mathbb{R}^+ \cup \infty$ eine Kürzeste-Wege-Funktion für u in G bzgl. Kostenfunktion f'.
Dann gilt für alle Knoten $v \in V$: $d'_u(v) = d_u(v) + d_s(u) - d_s(v)$

Beweis:

(a) $f'((u, v)) \geq 0$, weil für jede Kante $(u, v) \in E$ die Dreiecksungleichung $d_s(u) + f((u, v)) \geq d_s(v)$ gilt.

(b) Sei $p = (w_1, w_2, \ldots, w_k)$ ein Weg in G von Knoten $u = w_1$ nach Knoten $v = w_k$.
Sei K der Kostenwert von p bzgl. Kostenfunktion f.
Sei K' der Kostenwert von p bzgl. Kostenfunktion f'.

$$\begin{aligned} K' &= K + d_s(w_1) - d_s(w_2) + d_s(w_2) - d_s(w_3) + \ldots \\ &\qquad \ldots + d_s(w_{k-1}) - d_s(w_k) \\ &= K + d_s(w_1) - d_s(w_k) \\ &= K + d_s(u) - d_s(v). \end{aligned}$$

Somit ist ein Weg p von u nach v genau dann ein kürzester Weg bzgl. Kostenfunktion f, wenn er ein kürzester Weg bzgl. Kostenfunktion f' ist.

Betrachte zwei Wege p_1, p_2 mit Kostenwerten K_1, K_2 bzgl. Kostenfunktion f und Kostenwerten K_1', K_2' bzgl. Kostenfunktion f'.
Dann ist
$K_1' = K_1 + d_s(u) - d_s(v)$ und
$K_2' = K_2 + d_s(u) - d_s(v)$.
Also ist $K_1' - K_1 = K_2' - K_2$ und somit $K_1 < K_2 \Leftrightarrow K_1' < K_2'$. $\square$

Da G mit Kostenfunktion f' ein Distanzgraph ist, kann das Kürzeste-Wege-Problem für alle Knotenpaare in Zeit $O(|V| \cdot (|E| + |V| \cdot \log(|V|)))$ berechnet werden (starte Dijkstras Algorithmus mit jedem Knoten).

8.3 Minimale spannende Bäume

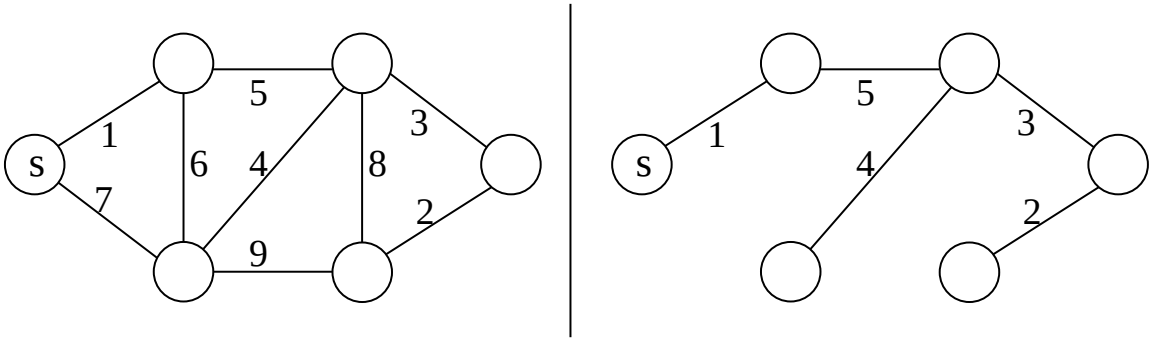
Gegeben: Ungerichteter, kantenberwerteter, zusammenhängender Graph $G = (V, E)$ mit Kantengewichtung $c : E \to \mathbb{R}$.
Gesucht: Spannender Baum $G' = (V, E')$ für G mit

$$\sum_{e \in E'} c(e)$$

minimal.

Es kann exponentiell viele Spannbäume geben.

Beispiel:



Trotzdem gibt es sehr effiziente Algorithmen für die Berechnung minimaler Spannbäume mit Laufzeiten aus $O(|E| + |V| \cdot \log(|V|))$.

Es geht sogar noch schneller mit Laufzeiten aus $O(|V|+|E|\cdot\alpha(|E|,|V|))$ (Fredman, Tarjan; 1987), wobei $\alpha(m,n)$ die inverse Ackermann-Funktion ist.

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

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

$A(i,j)$ besitzt ein explosives Wachstum, $\alpha(m,n)$ dagegen wächst minimal.

$\alpha(m,n) \leq 6$, falls m,n die Anzahl der Atome im Universum ist.

Greedy-Methoden zur Bestimmung minimaler Spannbäume:

Benutze zwei Farben um die Kanten zu färben (grün für die Kanten des minimalen Spannbaumes, rot für die Kanten, die nicht zum gesuchten Spannbaum gehören sollen). In jedem Schritt wird genau eine noch ungefärbte Kante grün oder rot gefärbt. Einmal getroffene Entscheidungen werden nicht mehr zurückgenommen.

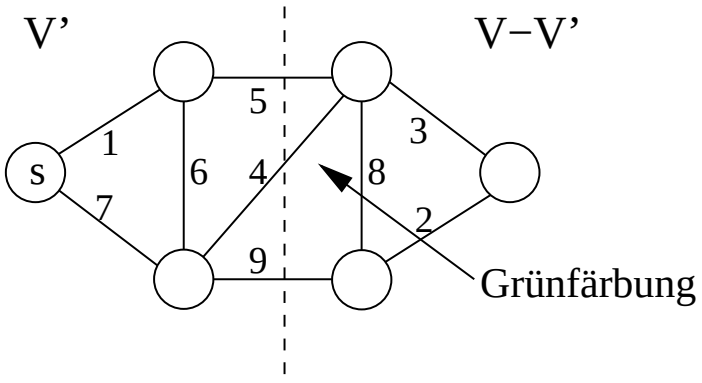
Grüne Kantenregel

Wähle eine Knotenmenge $V' \subseteq V$, so dass

1. es keine grüne Kante $\{u', u\} \in E$ gibt, mit $u' \in V'$ und $u \in V - V'$ und
2. es noch eine ungefärbte Kante $\{v', v\} \in E$ gibt, mit $v' \in V'$ und $v \in V - V'$.

Färbe eine der ungefärbten Kanten $\{v', v\} \in E$ mit $v' \in V'$ und $v \in V - V'$ und minimalen Kosten grün.

Beispiel:



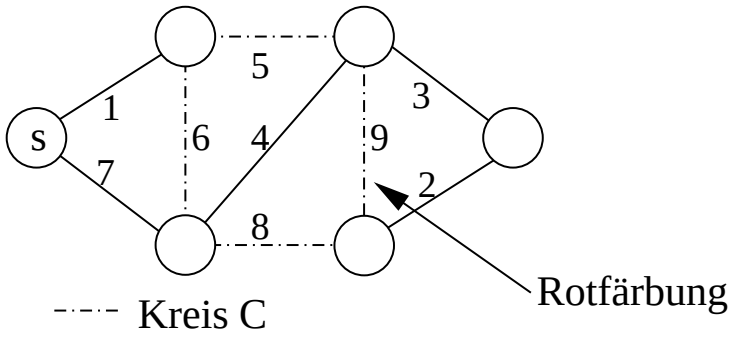
Rote Kantenregel

Wähle einen Kreis C in G, so dass

1. es noch keine rote Kante in C gibt und
2. es noch eine ungefärbte Kante in C gibt.

Färbe eine der ungefärbten Kanten in C mit maximalen Kosten rot.

Beispiel:



Invariante für Greedy-Algorithmen:

Nach jedem Färbungsschritt gilt: Es gibt einen minimalen Spannbaum, der alle grünen und keine der roten Kanten enthält.

Beweis: Induktiv: Wenn es noch keine grünen und roten Kanten gibt, gilt die Invariante.
Sei $T = (V, E_T)$ ein minimalen Spannbaum für $G = (V, E)$, der alle bisher grün gefärbten aber keine der bisher rot gefärbten Kanten enthält.

1. Grünfärbung einer ungefärbten Kante $e = \{u', u\} \in E$ mit $u' \in V'$ und $u \in V - V'$.

   Falls $e \in E_T$, dann gilt nach der Grünfärbung weiterhin die Invariante.

   Falls $e \notin E_T$, dann sei $e' = \{v', v\} \in E'$ eine der Kanten aus E_T mit $v' \in V'$ und $v \in V - V'$, die mit e auf einem Kreis liegt. e' kann nicht grün sein, weil es keine grünen Kanten $\{v', v\}$ mit $v' \in V'$ und $v \in V - V'$ gibt. e' kann nicht rot sein, da T keine roten Kanten enthält. Also ist e' ungefärbt.

   Da $G' = (V, E')$ ein minimaler Spannbaum ist, folgt $c(e') \leq c(e)$. Aus der Voraussetzung für die Anwendung der grünen Regel folgt $c(e) \leq c(e')$. Also gilt $c(e) = c(e')$.

   Somit ist $(V, E_T - \{e'\} \cup \{e\})$ ein minimaler Spannbaum, der die Invariante erfüllt.

2. Rotfärbung einer ungefärbten Kante $e = \{u', u\} \in E$ auf einem Kreis C.

   Falls $e \notin E_T$, dann gilt nach der Rotfärbung weiterhin die Invariante.

Falls $e \in E_T$, dann zerfällt der Baum T nach dem Entfernen von e in zwei Bäume. Die Knotenmengen dieser Bäume definieren einen Schnitt $V' \subseteq V$, so dass alle Kanten $\{w', w\} \in E$ mit $w' \in V'$ und $w \in V - V'$ nicht in E_T sind.

Sei $e' = \{u', u\}$ eine Kante aus dem Kreis C mit $u' \in V'$ und $u \in V - V'$. e' kann nicht grün sein, weil E_T alle grünen Kanten enthält. e' kann nicht rot sein, weil C keine roten Kanten enthält. Also ist e' ungefärbt.

Da $G' = (V, E')$ ein minimaler Spannbaum ist, folgt $c(e') \geq c(e)$. Aus der Voraussetzung für die Anwendung der roten Regel folgt $c(e') \leq c(e)$. Also gilt $c(e') = c(e)$.

Somit ist $(V, E_T - \{e\} \cup \{e'\})$ ein minimaler Spannbaum, der die Invariante erfüllt.

Satz: Solange noch eine Kante e ungefärbt ist, läßt sich eine der beiden Regeln anwenden.

Beweis: Wenn e zwei grüne Bäume verbindet, ist die grüne Regel anwendbar (jedoch nicht unbedingt auf Kante e).
Wenn e zwei Knoten in demselben grünen Baum verbindet, ist die rote Regel (auf Kante e) anwendbar. □

Idee von Prim (1956)

Wähle einen Startknoten s.
Wende die grüne Regel auf den Schnitt an, der die Knoten im bereits aufgebauten grünen Baum mit den übrigen Knoten verbindet.
Nach $|V| - 1$ Anwendungen der grünen Regel ist der grüne Baum ein minimaler Spannbaum.

Idee von Kruskal (1956)

Sortiere die Kanten nach aufsteigenden Kosten und färbe sie in dieser Reihenfolge wie folgt: Falls eine Kante e zwei grüne Bäume verbindet, färbe e grün, sonst rot.

Prims Algorithmus:

```
(1)  Prim(s)
(2)  {
(3)      for all u ∈ V {
(4)          d[u] := ∞;
(5)          markiere u als „nicht besucht";
```

```
(6)          Vorgänger(u) := undefiniert; }
(7)      d[s] := 0;
(8)      markiere s als „besucht";
(9)      Vorgänger(s) := s;
(10)     initialisiere eine Priority-Queue R;
(11)     for all {s, w} ∈ E {
(12)          d[w] := c({s, w});
(13)          Vorgänger(w) := s;
(14)          insert(R, w); }
(15)     while (R ≠ ∅) {
(16)          v := extract-min(R);
(17)          markiere v als „besucht";
(18)          färbe die Kante {Vorgänger(v), v} grün;
(19)          for all (v, w) ∈ E {
(20)               if (w is als „nicht besucht" markiert
(21)               und c({v, w}) < d[w]) {
(22)                    d[w] := c({v, w});
(23)                    Vorgänger(w) := v;
(24)                    if (w ist nicht in R) { insert(R, w); };
(25)                    else { decrease(R, w, R[w]); } } } }
(26)  }
```

Laufzeit für Prims Algorithmus:

Jeder Knoten wird genau ein mal in R eingefügt und genau ein mal aus R entfernt. Die Anzahl der Decrease-Operationen ist somit aus $O(|E|)$.

Daraus folgt eine Laufzeit von $O(|E| + |V| \cdot \log(|V|))$ mit Fibonacci-Heaps als Priority-Queue. (Laufzeit für insert(D, k), im Worst-Case aus $O(1)$, für extract-min(D), amortisiert aus $O(\log(N))$, für decrease(D, k, y), amortisiert aus $O(1)$, siehe Kapitel 8 (Fibonacci-Heaps).)

Kruskals Algorithmus:

Benötigt wird eine Datenstruktur zur Verwaltung der grünen Bäume.

Die Datenstruktur Union-Find:

Gespeichert werden Mengen von Elementen, z.B.
$A = \{\underline{a_1}, a_2, a_3\}, B = \{\underline{b_1}, b_2, b_3, b_4\}, C = \{\underline{c_1}, c_2, c_3\}$.

Jede Menge M wird eindeutig durch ein kanonisches Element $x \in M$ repräsentiert, z.B. $\underline{a_1}, \underline{b_1}, \underline{c_1}$.

Operationen:

create(x):	Erzeugt für ein Element x eine Menge mit Element x.
union(x, y):	Vereinigt für zwei Repräsentanten x, y die beiden Mengen, die x und y repräsentieren.
find(x):	Liefert den Repräsentanten der Menge, die Element x enthält.

Beispiel:

find(a_2) = a_1 und find(b_3) = b_1

Zwei Elemente x, y sind genau dann in der gleichen Menge, wenn

$$\text{find}(x) = \text{find}(y).$$

Kruskals Algorithmus:

```
(1)  Kruskal(s)
(2)  {
(3)      ∀u ∈ V, erzeuge eine Menge mit Element u;
(4)      Sortiere die Kanten e nach aufsteigenden Kosten c(e) in einer Liste L;
(5)      for all {u, v} ∈ L in aufsteigender Reihenfolge {
(6)          u' := find(u);
(7)          v' := find(v);
(6)          if (u' ≠ v') {
(7)              union(u', v');
(8)              färbe {u, v} „grün"; }
(9)          else
(10)             färbe {u, v} „rot"; }
(11) }
```

Die Union-Find Datenstruktur kann sehr einfach so implementiert werden, dass jede Union-Operation auf zwei Repräsentanten $O(1)$ und jede Find-Operation $O(\log(|V|))$ viele Schritte benötigt.

$\Rightarrow$ Laufzeit für Kruskals Algorithmus: $O(|V| + |E| \cdot \log(|E|))$ (Die Zeit für die Kantensortierung dominiert die gesamte Laufzeit)

Realisierung der Union-Find Datenstruktur:

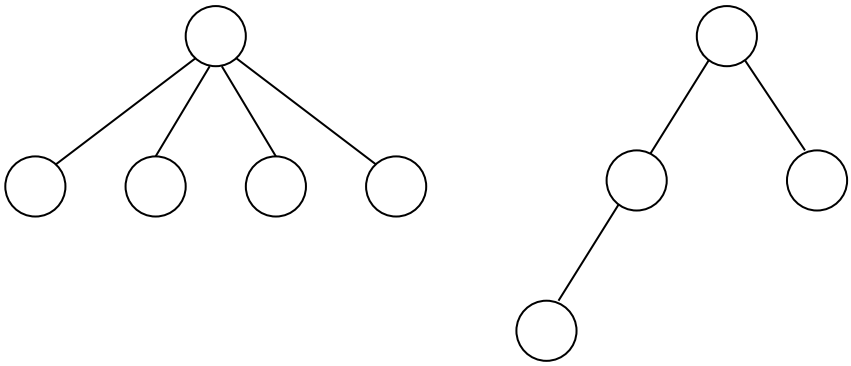
Die Elemente eine Menge werden in den Knoten eines Baumes beliebiger Ordnung gespeichert.

Der Repräsentant steht in der Wurzel des Baumes.

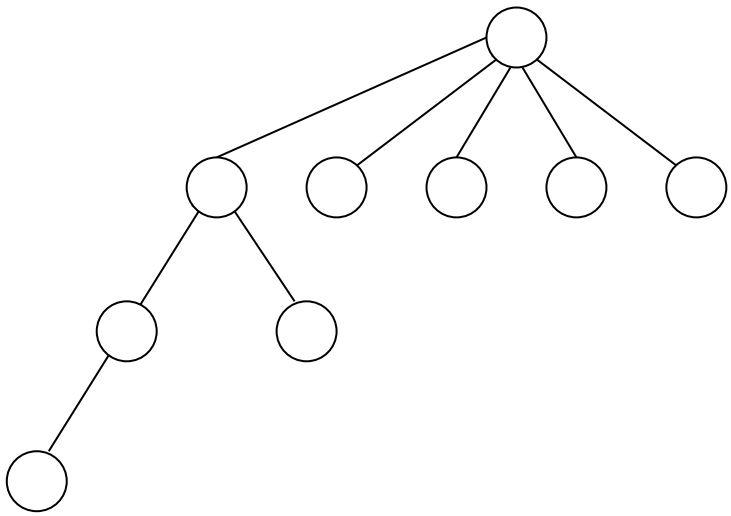
Bei der union(x, y)-Operatiopn wird die Wurzel des Baumes mit dem Element x (oder y) zum Sohn der Wurzel des Baumes mit Element y (bzw. x) gemacht.

Bei Vereinigung nach Größe (Höhe) wird die Wurzel des kleineren Baumes (des Baumes mit geringerer Höhe) zum Sohn der Wurzel des größeren Baumes (des Baumes mit größerer Höhe) gemacht. (An der Wurzel wird die Größe bzw. Höhe des Baumes gespeichert.)

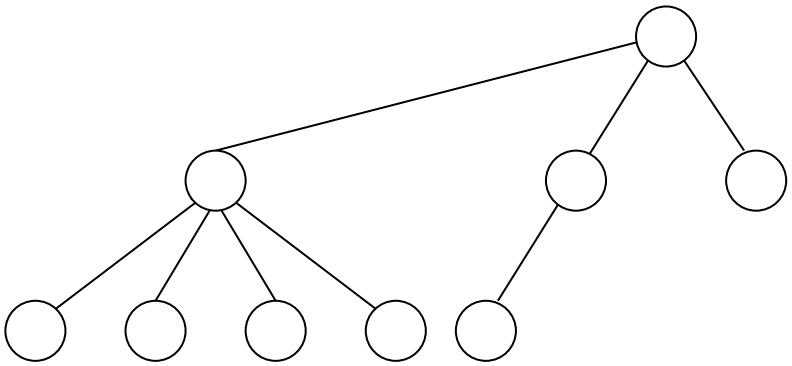
Beispiel:



Vereinigung nach Größe:



Vereinigung nach Höhe:



Lemma: Das Verfahren „Vereinigung nach Größe" läßt die folgende Eigenschaft der Bäume unverändert:

Ein Baum mit Höhe h hat wenigstens 2^h Knoten.

Beweis:
Induktiv:

Ein Baum der Höhe 0 hat mindestens $2^0 = 1$ Knoten.

Ein Baum der Höhe 1 hat mindestens $2^1 = 2$ Knoten.

Seien T_1 und T_2 zwei Bäume mit den Größen g_1 und g_2 und den Höhen h_1 und h_2. Sei $g_1 \geq g_2$ und T der durch Vereinigung nach Größe entstehende Baum mit Größe g und Höhe h.

1. Ist die neue Höhe $h = h_1$, dann hat T mindestens 2^h Knoten, weil T_1 bereits mindestens $2^{h_1} = 2^h$ Knoten hat.
2. Sei also $h > h_1$, dann ist $h = h_2 + 1$, da T_2 an den größeren Baum angehängt wurde.

   Nun gilt: $g_1 \geq g_2$ (Annahme) und $g_2 \geq 2^{h_2}$ (Induktionsvoraussetzung) und somit:

$$g = g_1 + g_2 \geq 2^{h_2} + 2^{h_2} = 2^{h_2+1} = 2^h.$$

Bemerkung:

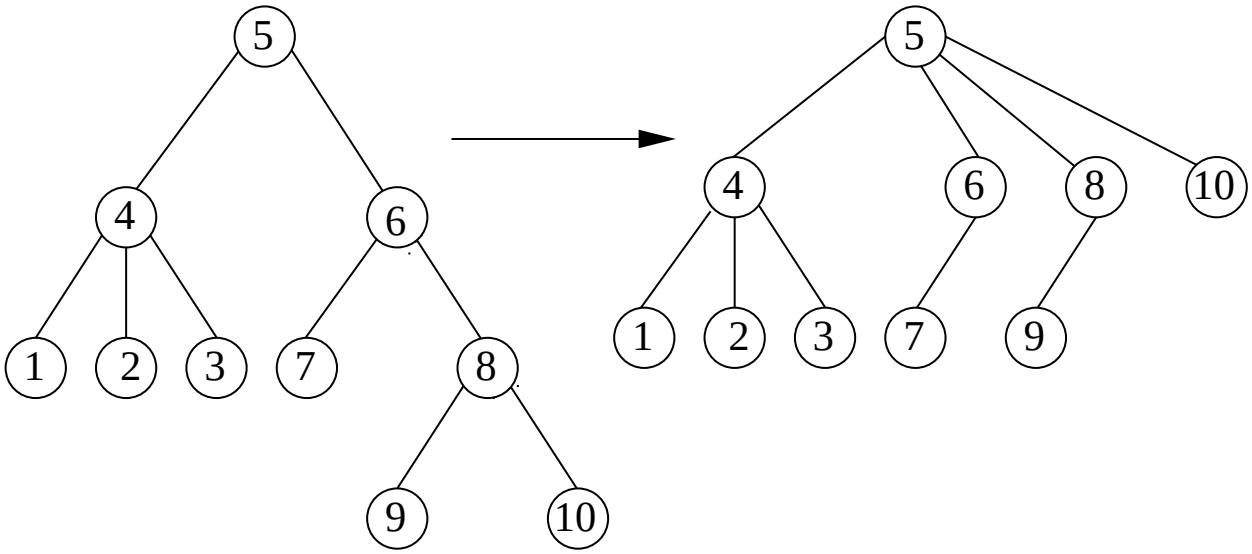
1. Das Lemma gilt auch für die Vereinigung nach Höhe.
2. Wird das Verfahren „Vereinigung nach Größe" angewendet, beginnend mit einer Folge von n Bäumen mit je genau einem Knoten, so haben alle entstehenden Bäume die Höhe $h \leq \log(|V|)$.

3. Zur Speicherung der Höhe wird weniger Platz benötigt als zur Speicherung der Größe.

Methode der Pfadverkürzung:

Bei Ausführung der find(x)-Operation läuft man von x zur Wurzel. Bei der Pfadverkürzung werden alle beim Hochlaufen besuchten Knoten direktan die Wurzel gehängt.

Beispiel:



Satz:

Sei n die Summe der Elemente, die in allen Mengen gespeichert sind.
Benutzt man bei der Ausführung der Find-Operation die Pfadverkürzung und bei der Ausführung der Union-Operationen die Vereinigung nach Größe/Höhe, so benötigt man zur Ausführung einer beliebigen Folge von $m \geq n$ Find- und Union-Operationen insgesamt $\Theta(m \cdot \alpha(m, n))$ Schritte (amortisierte Laufzeitabschätzung).

8.4 Netzwerkflussalgorithmen

Notationen:

Ein Netzwerk ist ein gerichteter Graph $G = (V, E)$ mit einer Kapazitätsfunktion

$$c : E \to \mathbb{R}_{\geq 0},$$

die jeder Kante eine nicht negative Kapazität zuordnet.

Eine Flussfunktion $f_{s,t}$ für $G = (V, E)$ bzgl. einer Quelle $s \in V$ und einer Senke $t \in V$ ist eine Abbildung

$$f_{s,t} : E \to \mathbb{R}_{\geq 0}$$

mit

- $\forall e \in E : 0 \leq f_{s,t}(e) \leq c(e)$
- $\forall v \in V - \{s, t\}$:

$$\sum_{e \in \text{in}(v)} f_{s,t}(e) = \sum_{e \in \text{out}(v)} f_{s,t}(e)$$

mit
$\text{in}(v) = \{(u, v) \in E \mid \text{ für ein } u \in V\}$
$\text{out}(v) = \{(v, w) \in E \mid \text{ für ein } w \in V\}$

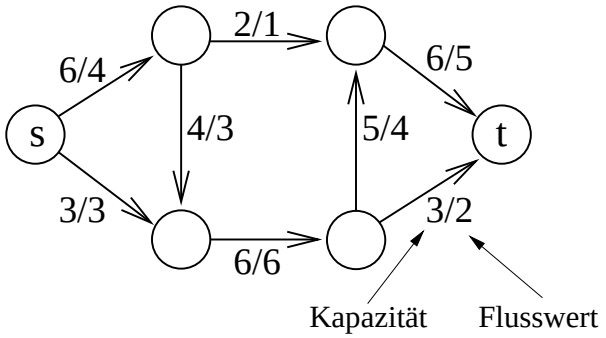
Der Fluss $F(f_{s,t})$ einer Flussfunktion $f_{s,t}$ für G bzgl. s und t ist

$$F(f_{s,t}) = \sum_{e \in \text{out(s)}} f_{s,t}(e) - \sum_{e \in \text{in}(s)} f_{s,t}(e).$$

Das Netzwerkflussproblem:

Gegeben: Ein Netzwerk $G = (V, E)$ mit Kapazitätsfunktion $c : E \to \mathbb{R}_{\geq 0}$, eine Quelle $s \in V$ und eine Senke $t \in V$.
Gesucht ist eine Flussfunktion $f_{s,t}$ für G bzgl. s und t mit einem maximalen Fluss $F(f_{s,t})$.

Beispiel:



Beobachtung:

Der Fluss einer Flussfunktion kann an jedem s, t-Schnitt gemessen werden.

Sei $S \subseteq V$, $\overline{S} = V - S$,

$$E(S, \overline{S}) = \{(u, v) \mid u \in S, v \in \overline{S}\}$$

und

$$E(\overline{S}, S) = \{(u, v) \mid u \in \overline{S}, v \in S\}.$$

Ein Paar $S, \overline{S}$ mit $s \in S$ und $t \in \overline{S}$ ist ein s,t-Schitt.

Lemma: Für jedes $S \subseteq V$, $\overline{S} = V - S$ mit $s \in S$ und $t \in V - S$ gilt:

$$F(f_{s,t}) = \sum_{e \in E(S,\overline{S})} f_{s,t}(e) - \sum_{e \in E(\overline{S},S)} f_{s,t}(e)$$

Beweis: Starte mit $S = \{s\}$ und nehme schrittweise in beliebiger Reihenfolge die gewünschten Knoten v zu S hinzu. Da $\forall v \in V - \{s,t\} : \sum_{e \in \text{in}(v)} f_{s,t}(e) = \sum_{e \in \text{out}(v)} f_{s,t}(e)$, bleibt

$$\sum_{e \in \text{out(s)}} f_{s,t}(e) - \sum_{e \in \text{in}(s)} f_{s,t}(e) = \sum_{e \in E(S,\overline{S})} f_{s,t}(e) - \sum_{e \in E(\overline{S},S)} f_{s,t}(e)$$

Sei
$a_1 = \sum_{e \in E(S,\{v\})} f_{s,t}(e)$,
$a_2 = \sum_{e \in E(\overline{S},\{v\})} f_{s,t}(e)$,
$b_1 = \sum_{e \in E(\{v\},\overline{S})} f_{s,t}(e)$ und
$b_2 = \sum_{e \in E(\{v\},S)} f_{s,t}(e)$.

Da $a_1 + a_2 = b_1 + b_2$ gilt nach der Aufnahme von v

$$F_{f_{s,t}} + b_1 + b_2 - (a_1 + a_2) \;=\; F_{f_{s,t}}.$$

□

Beobachtungen

1. Sei $c(S) = \sum_{e \in E(S,\overline{S})} c(e)$,
   dann gilt für jedes S mit $s \in S$ und $t \notin S : F(f_{s,t}) \leq c(S)$.
2. Min-Cut Max-Flow Theorem:
   Sei $F_{s,t,\max} = \max_{f_{s,t}} F(f_{s,t})$ und $C_{s,t,\min} = \min_S c(S)$.
   Falls $F(f_{s,t}) = c(S)$, dann ist $F(f_{s,t}) = F_{s,t,\max}$ und $c(S) = C_{s,t,\min}$.

Ein Pfad in einem Netzwerk $G = (V, E)$ ist eine Knotenfolge $P = u_1, \ldots, u_k$, so dass für jedes Knotenpaar u_i, u_{i+1}, $1 \leq i \leq k-1$, entweder (u_i, u_{i+1}) oder (u_{i+1}, u_i) eine gerichtete Kante in G ist.

(u_i, u_{i+1}) ist eine Vorwärtskante und (u_{i+1}, u_i) eine Rückwärtskante in Pfad P.

Idee der meisten Flussalgorithmen:

1. Starte mit einer Flussfunktion $f_{s,t}$, z.B. $f_{s,t}(e) = 0 \ \ \forall e \in E$.
2. Suche einen Pfad $P = u_1, \ldots, u_k$ mit $u_1 = s$ und $u_k = t$, so dass
   (a) für alle Vorwärtskanten $e = (u_i, u_{i+1}) : f_{s,t}(e) < c(e)$
   [Sei $\Delta(e) = c(e) - f_{s,t}(e)$]
   (b) und für alle Rückwärtskanten $e = (u_{i+1}, u_i) : f(e) > 0$
   [Sei $\Delta(e) = f_{s,t}(e)$]

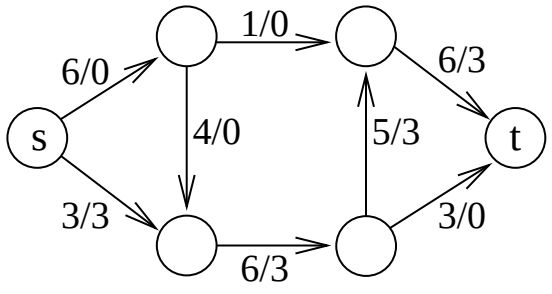
   Sei $\Delta(P)$ das kleinste $\Delta(e)$ über alle Kanten e im Pfad.
3. Erhöhe den Fluss um $\Delta(P)$, indem für alle Vorwärtskanten der Fluss $f(e)$ um $\Delta(P)$ erhöht wird, und für alle Rückwärtskanten e der Fluss $f(e)$ um $\Delta(P)$ erniedrigt wird.
4. Wiederhole die Schritte 2. und 3. solange es noch einen Pfad P mit $\Delta(P) > 0$ gibt.

Ein Pfad P mit $\Delta(P) > 0$ ist ein zunehmender Pfad.

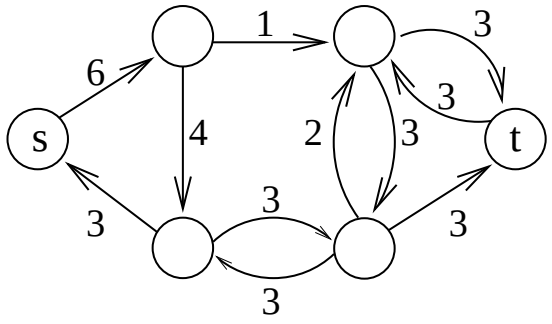
Sei $G_R(f_{s,t}) = (V, E_R)$ der Restgraph zu $f_{s,t}$, der alle noch möglichen Flussvergrößerungen beschreibt.

Für jede Kante $e \in E$ von u nach v gibt es im Restgraphen
eine Kante von u nach v mit Gewicht $c(e) - f_{s,t}(e)$, falls $c(e) > f_{s,t}(e)$, und
eine Kante von v nach u mit Gewicht $f_{s,t}(e)$, falls $f_{s,t}(e) > 0$.

Beispiel:



Restgraph:



Bemerkung:

Jeder Weg von s nach t im Restgraphen $G_R(f_{s,t})$ ist ein zunehmender Pfad in G.

Satz: Der Fluss einer Flussfunktion $f_{s,t}$ ist genau dann maximal, wenn es in $G = (V, E)$ keinen zunehmenden Pfad mehr gibt.

Beweis: 1.) Wenn es noch einen zunehmenden Pfad in G gibt, dann kann der Fluss entlang des Pfades vergrößert werden.

2.) Angenommen, es gibt keinen zunehmenden Pfad in G.
Sei S die Menge der Knoten, die im Restgraphen $G_R(f_{s,t})$ von s aus erreichbar sind.

Nun gilt:

$s \in S$,

$t \notin S$,

für alle Kanten $e = (u, v) \in E$ mit $u \in S, v \in \overline{S}$ ist $f_{s,t}(e) = c(e)$, und

für alle Kanten $e = (v, u) \in E$ mit $u \in S, v \in \overline{S}$ ist $f_{s,t}(e) = 0$.

Aus 3. und 4. folgt, dass $F(f_{s,t}) = c(S)$.

Da immer $F_{max} \leq c(S)$ gilt, folgt nun $F(f) = F_{max}$. $\square$

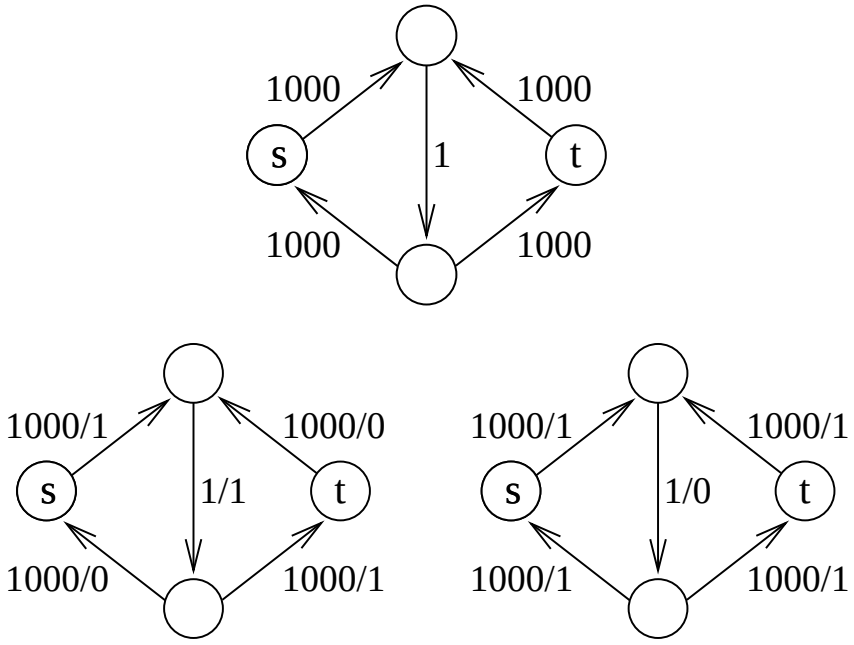
Idee von Ford-Fulkerson (1956)

Suche einen beliebigen zunehmenden Pfad P und erhöhe den Fluss entlang des Pfades um $\Delta(P)$.

Bemerkung:

1. Leider terminiert der Algorithmus nicht immer für irrationale Kapazitäten (ohne Beweis).
2. Er konvergiert für irrationale Kapazitäten nicht einmal unbedingt gegen F_{max} (ohne Beweis).
3. Für ganzzahlige Kapazitäten ist die Anzahl der Flussvergrößerungen durch F_{max} beschränkt.

Beispiel:



usw...

Verbesserung:

Suche geeignete zunehmende Pfade!

Erste Idee von Karp und Edmonds:

Wähle immer einen zunehmenden Pfad mit maximaler Flussvergrößerung.

Sei $f_{s,t}$ eine Flussfunktion für G bzgl. s und t.

Bemerkung:

Jede Flussfunktion $f_{s,t}$ kann mit $\leq m = |E|$ geschickt gewählten zunehmenden Pfaden ohne Rückwärtskanten konstruiert werden, so dass jeder Pfad mindestens einer Kante e den Flusswert $f_{s,t}(e)$ zuordnet.

$\Rightarrow$ Man kann mit $\leq m$ geschickt gewählten Flussvergrößerungen eine Flussfunktion mit maximalem Fluss konstruieren.

$\Rightarrow$ Ist eine Flussfunktion $f_{s,t}$ vorgegeben, so kann man mit $\leq m$ geschickt gewählten Flussvergrößerungen eine Flussfunktion $f'_{s,t}$ mit $F(f'_{s,t}) = F_{max} - F(f_{s,t})$ konstruieren.

$\Rightarrow$ Der Fluss einer Flussfunktion $f_{s,t}$ wird bei einer maximalen Flussvergrößerung um mindestens $\frac{F_{\max} - F(f_{s,t})}{m}$ vergrößert.

$\Rightarrow$ Die Flüsse einer maximalen Flussvergrößerung sind somit

$$
\begin{aligned}
&F_0 \\
&F_1 \geq F_0 + \tfrac{F_{\max} - F_0}{m} \\
&F_2 \geq F_1 + \tfrac{F_{\max} - F_1}{m} \\
&F_3 \geq F_2 + \tfrac{F_{\max} - F_2}{m} \\
&\quad \vdots
\end{aligned}
$$

$\Rightarrow$ Nach höchstens $2 \cdot m$ maximalen Flussvergrößerungen ist die Flussvergrößerung auf mindestens die Hälfte gesunken (Übungsaufgabe).

$\Rightarrow$ Anzahl der Flussvergrößerungen ist $O(|E| \cdot \log(c_{max}))$, wobei c_{max} die maximale Kapazität in G ist, also eine obere Schranke für die maximale Flussvergrößerung mit einem zunehmenden Pfad ist.

$\Rightarrow$ mit Dijkstras Kürzeste-Wege-Algorithmus können maximale zunehmende Pfade in Zeit $O(|E| + |V| \cdot \log(|V|))$ bestimmt werden.

$\Rightarrow$ gesamte Laufzeit: $O((|E| + |V| \cdot \log(|V|)) \cdot |E| \cdot \log(c_{max}))$

Zweite Idee von Karp und Edmonds:

Wähle immer einen zunehmenden Pfad mit einer minimalen Anzahl von Kanten.

Lemma: Werden immer nur Pfade mit minimaler Anzahl von Kanten zur Flussvergrößerung herangezogen (sogenannte kürzeste Pfade), so erhöht sich die Anzahl der Kanten auf den kürzesten Pfaden nach höchstens m Iterationen mindestens um 1.

Beweisidee: Angenommen es gibt keine zunehmenden Pfade der Länge $< k$, aber einen zunehmenden Pfade der Länge k. Durch die Anwendung eines zunehmenden Pfades der Länge k erhält mindestens eine Kante (u, v) entweder den Fluss $c((u, v))$, falls die Kante im Pfad in Vorwärtsrichtung war, oder den Fluss 0, falls

die Kante im Pfad in Rückwärtskante war. Diese Kante wird anschließend in keinem weiteren Pfad der länge $\leq k$ in umgekehrter Richtung verwendet. $\square$

$\Rightarrow$ Damit erhalten wir $(|V|-1) \cdot |E|$ Iterationen.

Kürzeste Pfade können mit Breitensuche in Zeit $O(|E|)$ bestimmt werden.

$\Rightarrow$ gesamte Laufzeit: $O(|E|^2 \cdot |V|)$

Idee von Dinic (1970):

Betrachte alle kürzesten Pfade mit gleicher Anzahl von Kanten in einer Runde.

1. Bilde den Niveaugraphen $G_N(f_{s,t}) = (V, E_N)$

   $G_N(f_{s,t})$ ist der Teilgraph vom Restgraphen $G_R(f_{s,t})$, der nur die kürzesten Wege von s enthält.

   Der Niveaugraph kann in Zeit $O(|E|)$ aufgebaut werden. (Breitensuche)

2. Bestimme einen Flussfunktion $f'_{s,t}$ für den Niveaugraphen $G_N(f_{s,t})$, die nicht durch einen zunehmender Pfad von s nach t mit ausschließlich Vorwärtskanten erhöhbar ist und addiere $f'_{s,t}$ zu $f_{s,t}$ hinzu. Dies ist ein sogenannter blockierender Fluss für den Niveaugraphen.

3. Wiederhole 1. und 2. solange, bis $f_{s,t}$ maximal ist (d.h. bis es keinen Weg von s nach t in $G_N(f_{s,t})$ gibt).

Da nach jeder Iteration die Anzahl der Kanten in den kürzesten Wegen sich um mindestens 1 erhöht, wird nach höchstens $(|V|-1)$ Iterationen eine Flussfunktion mit maximalem Fluss berechnet. Bei jeder Iteration werden alle kürzesten Wege von s nach t gleichzeitig zu einer Flussvergrößerung herangezogen.

Aufbau des Niveaugraphen:

Partitioniere V in $V_0, V_1, \ldots, V_k$ mit $V_i \cap V_j = \emptyset$ für $i \neq j$ und $V = \bigcup_j V_j$:

$V_0 := \{s\}$;

$i := 0$;

solange $(t \notin V_i)$, wiederhole {

$i := i + 1$;

$V_i := \{\}$;

markiere alle Knoten in V_{i-1} als „besucht";

betrachte für jeden Knoten $u \in V_{i-1}$ alle Kanten $e = (u, v) \in E$ {

falls v noch nicht besucht ist und $(f(e) < c(e))$, dann füge v zu V_i hinzu;

betrachte für jeden Knoten $u \in V_{i-1}$ alle Kanten $e = (v, u) \in E$ {

falls v noch nicht besucht ist und $(f(e) > 0)$, dann füge v zu V_i hinzu}
}

Finden einer nicht vergrößerbaren Flussfunktion in $G_N(f_{s,t})$ in Zeit $O(|V| \cdot |E|)$:

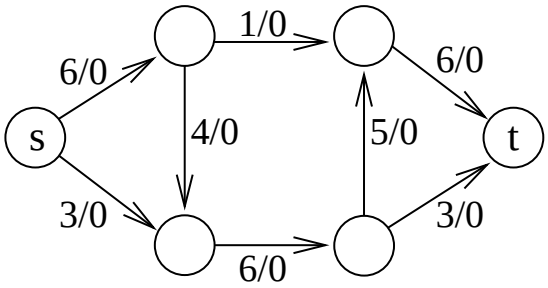
Starte eine Tiefensuche bei s bis Knoten t oder bis zu einem Knoten ohne auslaufende Kanten, das geht in Zeit $O(|V|)$.
Wenn Knoten t erreicht wird, ist ein zunehmender Pfad von s nach t gefunden. Verkleinere in diesem Fall alle Kapazitäten der Kanten von p um $\Delta(p)$, geht in Zeit $O(|V|)$.
Dabei erhält mindestens eine Kante die Kapazität 0. Entferne diese Kanten aus dem Niveaugraphen und starte die Tiefensuche erneut. Wenn t nicht erreicht wird, entferne die letzte Kante des Weges aus dem Niveaugraphen.

Laufzeit: $O(|E| \cdot |V|)$.

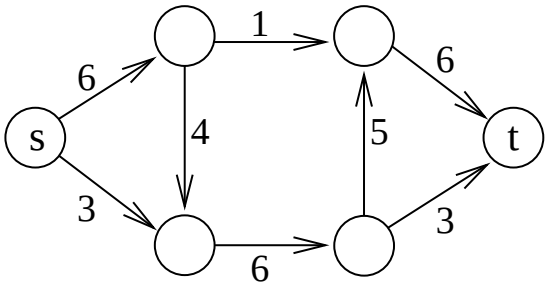
Aus $|V| - 1$ Iterationen folgt eine gesamte Laufzeit für Dinics Algorithmus von $O(|E| \cdot |V|^2)$.

Beispiel:

(a)

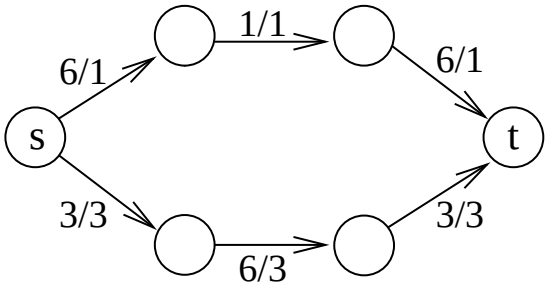


Restgraph:

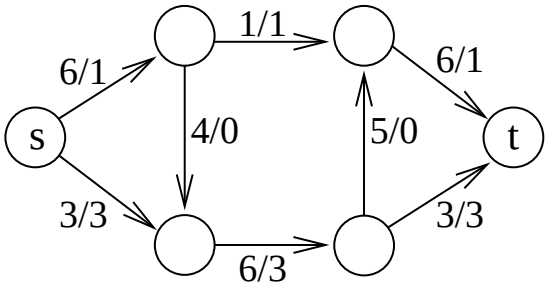


Niveaugraph:

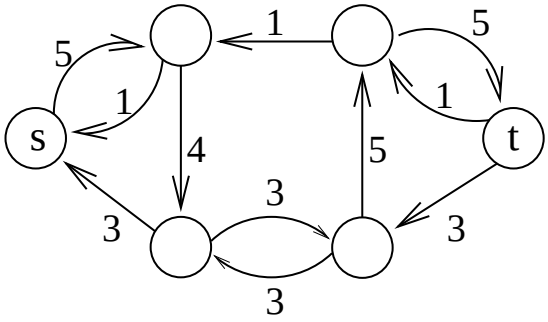
blockierender Fluss



(b)

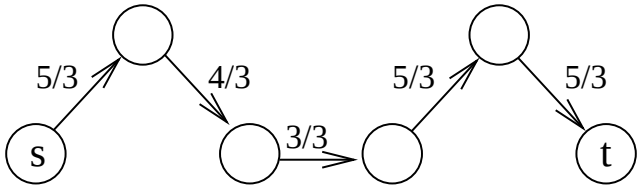


Restgraph:

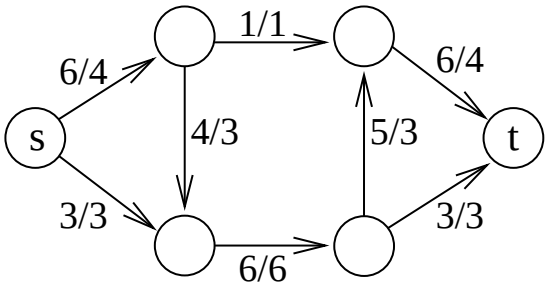


Niveaugraph:

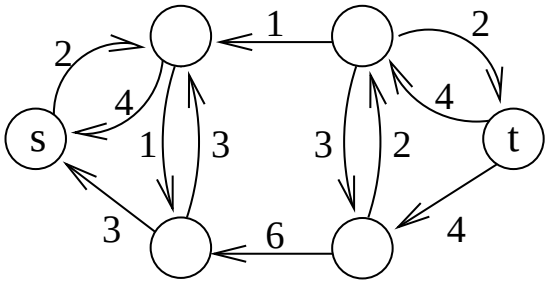
blockierender Fluss



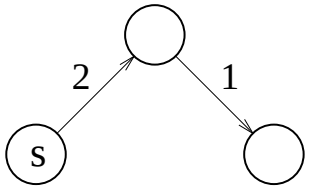
(c)



Restgraph:



Niveaugraph:



Spezialfall: (Even, Tarjan; 1975)

Wenn alle Kapazitäten 1 sind und jeder Knoten außer s und t genau eine einlaufende oder genau eine auslaufende Kante hat, können blockierende Flüsse im Niveaugraphen in Zeit $O(|E|)$ gefunden werden.

Beweisidee: Verfeinere die Tiefensuche von s auf dem Niveaugraphen, so dass bei einem Knoten ohne weitere auslaufende Kanten, die letzte Kante zwar gelöscht aber die Tiefensuche am Startknoten dieser Kante fortgesetzt wird. Erreicht die Suche Knoten t, so ist ein Weg p von s nach t gefunden. Nun wird eine Flußvergrößerung um $\Delta(p)$ entlang des Weges p durchgeführt und alle Kanten des Weges p aus dem Niveaugraphen entfernt. Anschließend wird eine neue Tiefensuche bei s gestartet.

Laufzeit: $O(|E|)$.

In diesem Spezialfall erfolgen höchstens $O(\sqrt{|V|})$ Iterationen.

Beweisidee: Ohne Beschränkung der Allgemeinheit gibt es keine Kante zwischen s und t.

Beobachtungen:

1. Jeder zunehmende Pfad erhöht den Fluss genau um den Wert 1.
2. Die Anzahl der knotendisjunkten Wege zwischen s und t entspricht genau dem maximalen Fluss $F_{\max}$ von s nach t.
3. Wenn l die Anzahl der inneren Knoten auf einem kürzesten zunehmenden Pfad von s nach t ist, so gilt
$$F_{\max} \leq \frac{|V|-2}{l}$$
bzw.
$$l \leq \frac{|V|-2}{F_{\max}}.$$

Angenommen $F_{\max} \leq \sqrt{|V|}$, dann ist die Anzahl der Iterationen mit Dinics Algorithmus $\leq \sqrt{|V|}$, da sich der berechnete Fluss nach jeder Iteration mindestens um den Wert 1 erhöht.

Sei nun $F_{\max} > \sqrt{|V|}$. Betrachte die i-te Iteration, so dass der bisher berechnete Fluss vor der i-ten Iteration $\leq F_{\max} - \sqrt{|V|}$ ist und nach der i-ten Iteration $> F_{\max} - \sqrt{|V|}$ ist. Der bisher berechnete Fluss kann also vor der i-ten Iteration mindestens noch um den Wert $\sqrt{|V|}$ und nach der i-ten Iteration höchstens noch um den Wert $\sqrt{|V|} - 1$ erhöht werden.

Für das Netzwerk vor der i-ten Iteration gilt somit
$$l \leq \frac{|V|-2}{\sqrt{|V|}}.$$

Das bedeutet, dass bisher höchstens $l - 1 \in O(\sqrt{|N|})$ Iterationen erfolgten, da die Längen der kürzesten Wege nach jeder Iteration um mindestens 1 wachsen.

Die Anzahl der Iterationen nach der i-ten Iteration kann aber nur $\sqrt{|V|} - 1$ sein, da der Fluss nach der i-ten Iteration höchstens noch um den Wert $\sqrt{|V|} - 1$ erhöht werden kann.

Die Anzahl der Iterationen ist als höchstens
$$(\frac{|V|-2}{\sqrt{|V|}} - 1) + 1 + (\sqrt{|V|} - 1).$$

$\Longrightarrow$ Laufzeit für Dinics Algorithmus im Spezialfall: $O(\sqrt{|V|} \cdot |E|)$

8.5 Anwendungen für Netzwerkflussalgorithmen

Matching-Probleme

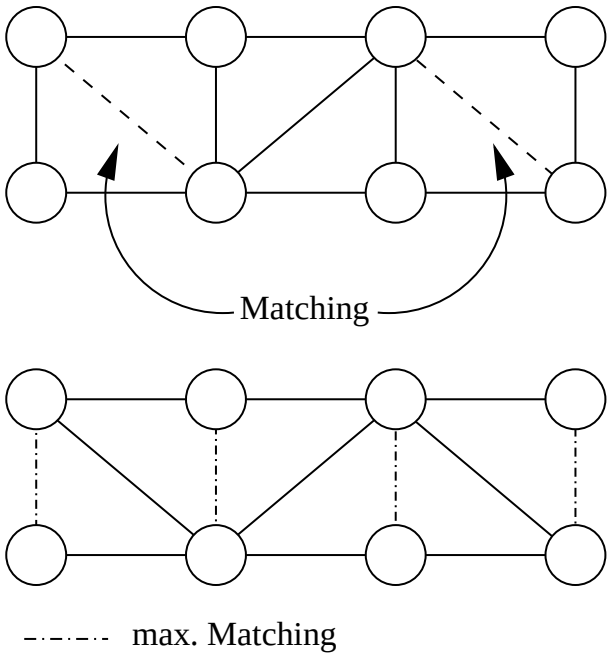
Definition: Sei $G = (V, E)$ ein ungereichteter Graph. Ein Matching ist eine Teilmenge $Z \subseteq E$, so dass keine zwei Kanten aus Z einen gemeinsamen Knoten haben.

Gegeben: Ein ungerichteter Graph $G = (V, E)$
Gesucht: Eine maximale Kantenmenge, die keine zwei inzidenten Kanten enthält.

Es wird in der Literatur unterschieden, zwischen einem Maximum-Matching (ein größtes Matching) und einem maximalen Matching (ein nicht vergrößerbares Matching).

Wir verwenden den Begriff maximales Matching für ein größtmögliches Matching und nicht für ein Matching, dass nicht durch Hinzunahme einer Kante vergrößert werden kann.

Beispiel:



Ein ungerichteter Graph $G = (V, E)$ ist bipartite, wenn sich die Knotenmenge V in zwei Mengen $V_1, V_2 \subseteq V$ zerlegen lässt, so dass $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$, und für alle $\{u, v\} \in E : u \in V_1, v \in V_2$ oder $u \in V_2, v \in V_1$.

Ein maximales Matching für bipartite Graphen kann mit einem Flussalgorithmus berechnet werden.

Konstruiere das Netzwerk $G' = (V', E')$ mit $V' = V \cup \{s, t\}$,

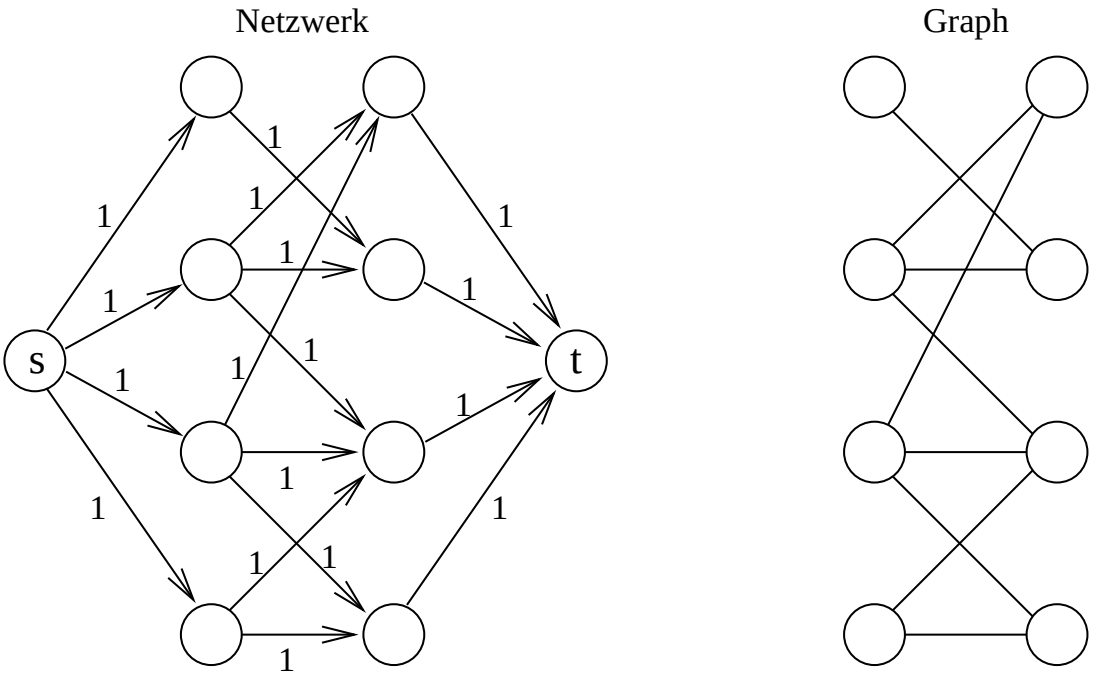
$$E' = \{s\} \times V_1 \cup \{(u, v) \mid \{u, v\} \in E, u \in V_1, v \in V_2\} \cup V_2 \times \{t\}$$

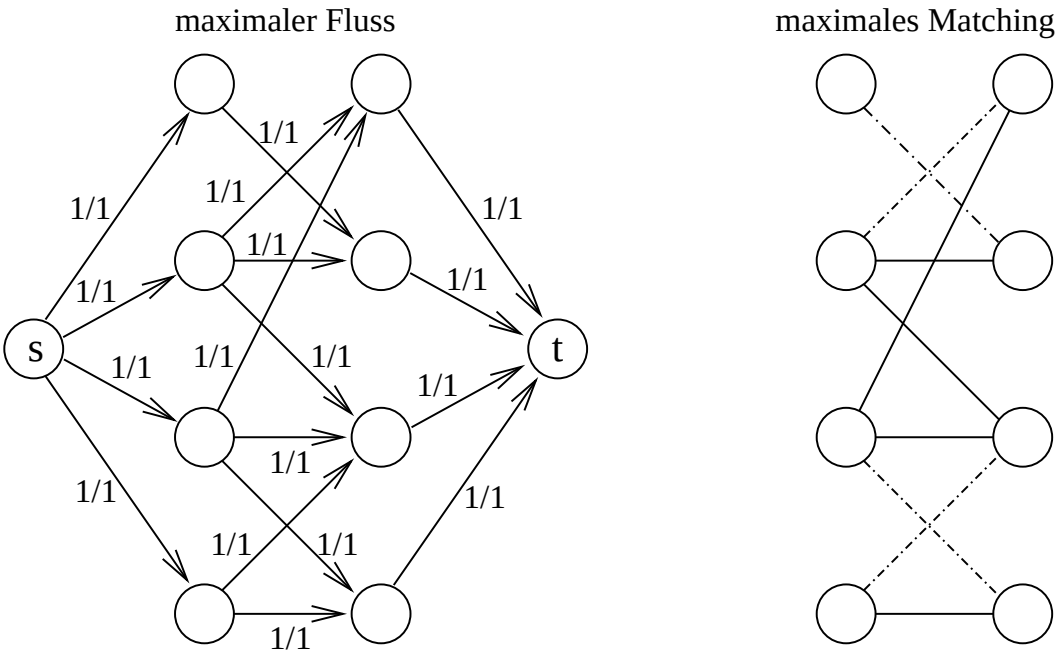
und $c : E \to \mathbb{R}_{\geq 0}$ mit $c(e) = 1$ für alle Kanten $e \in E'$.

Bestimme für G' einen maximalen Fluss durch zunehmende Pfade p mit $\Delta(p) = 1$. Jede Kante in G' hat also einen Flusswert, der entweder 0 oder 1 ist. Die Kanten $\{u, v\} \in E$ mit $f_{s,t}((u, v)) = 1$ bilden ein maximales Matching in G.

$\Rightarrow$ In bipartite Graphen kann ein maximales Matching in Zeit $O(|E| \cdot \sqrt{|V|})$ berechnet werden.

Beispiel:





Anwendung der Flusstheorie auf allgemeine Matching-Probleme

Zunehmende Pfade für Matching-Probleme:

Sei $Z \subseteq E$ ein Matching für Graph $G = (V, E)$.

Die Kanten in Z sind die gebundene Kanten, die übrigen sind die freien Kanten.

Ein Pfad, der abwechselnd aus freien und gebundenen Kanten besteht, ist ein alternierender Pfad.

Ein alternierender Pfad ist ein zunehmender alternierender Pfad, wenn beide Endkanten frei sind und beide Endknoten $u \neq v$ mit keiner gebundenen Kante inzident sind.

Sei P ein zunehmender alternierender Pfad für ein Matching $Z \subseteq E$. Das bestehende Matching Z kann vergrößert werden, indem alle gebundenen Kanten in P aus Z entfernt werden und alle freien Kanten in P zu Z hinzugefügt werden.

Lemma: Durch fortlaufendes Finden zunehmender alternierender Pfade kann ein maximales Matching berechnet werden.

Beweisidee:

Sei Z eine beliebige Zuordnung.
Sei $Z_{\max}$ eine maximale Zuordnung.
Sei $k = |Z_{\max}| - |Z|$.

Betrachte die symmetrische Differenz $Z_{sym} = (Z_{\max} - Z) \cup (Z - Z_{\max})$.

Jeder Knoten in G ist mit höchstens zwei Knoten aus Z_{sym} inzident. (eine Kante aus $Z_{\max}$ und eine Kante aus Z).

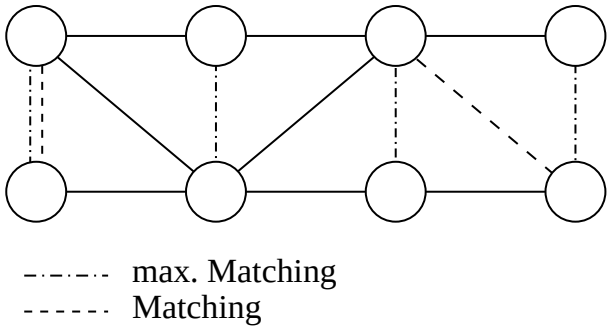
Jeder Zyklus hat eine gerade Anzahl von Kanten, da die Kanten aus $Z_{\max}$ und Z sich abwechseln.

$\Rightarrow$ Es gibt nur Zyklen gerader Länge.

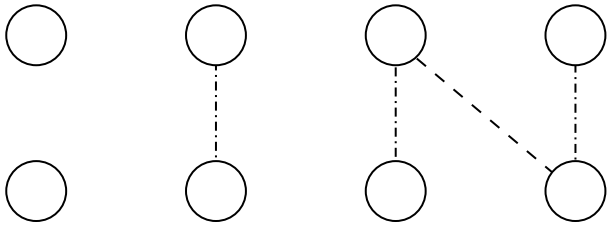
Da $Z_{\max}$ k Kanten mehr enthält als Z und in Z_{sym} alle Kanten aus $Z_{\max} \cup Z$ sind, die nicht in beiden Mengen sind, ist die Anzahl der aus $Z_{\max}$ stammenden Kanten um k größer als die Anzahl der aus Z stammenden Kanten.

$\Rightarrow$ Es gibt mindestens k Wege, die mit einer Kante aus Z anfangen und enden.

Beispiel:



symmetrische Differenz:



Bemerkung:

In bipartiten Graphen können zunehmende alternierende Pfade einfach mit einer Breitensuche berechnet werden.

Für allgemeine Graphen gibt es ebenfalls Machting-Algorithmen mit einer Laufzeit aus $O(|E| \cdot \sqrt{|V|})$, siehe zum Beispiel: T. Ottmann und P. Widmeyer, Kapitel 8.8.

Wege- und Zusammenhangsprobleme

Kantendisjunkte s,t-Wege

Gegeben: Ein gerichteter Graph $G = (V, E)$, ein Startknoten $s \in V$ und ein Zielknoten $t \in V$.
Gesucht ist die maximale Anzahl an Wegen von s nach t, die paarweise keine gemeinsame Kante enthalten.

Setze $c(e) = 1$ für alle Kanten $e \in$E.

Der maximale Fluss von s nach t entspricht der maximalen Anzahl von kantendisjunkten Wegen von s nach t.

Knotendisjunkte s,t-Wege

Gegeben: Ein gerichteter Graph $G = (V, E)$, ein Startknoten $s \in V$ und ein Zielknoten $t \in V$.
Gesucht ist die maximale Anzahl an Wegen von s nach t, die paarweise keinen gemeinsamen Knoten enthalten außer s und t.

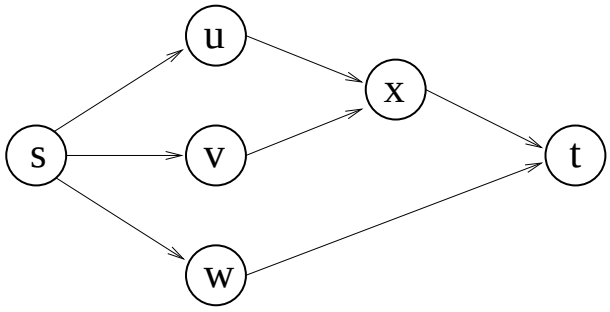
Ersetze jeden Knoten u durch zwei Knoten u_1, u_2 und eine Kante (u_1, u_2), und ersetze jede Kante (v, w) durch die Kante (v_2, w_1). Sei G' der resultierende Graph.

Die maximale Anzahl von knotendisjunkten Wegen von s nach t in G entspricht der maximalen Anzahl von kantendisjunkten Wegen von s_2 nach t_1 in G'.

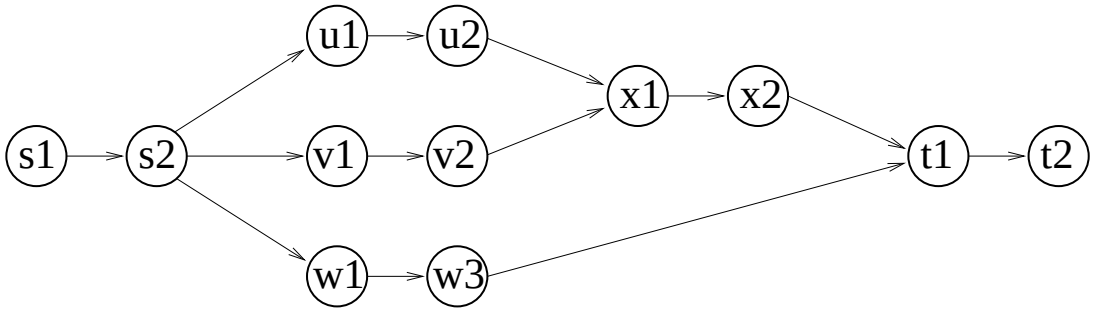
$\Rightarrow$ Die maximale Anzahl der knotendisjunkten Wege von s nach t kann in Zeit $O(|E| \cdot \sqrt{|V|})$ berechnet werden.

Beispiel:

Graph G :



Graph G' :



k-facher Zusammenhang (Zusammenhangszahl)

Gegeben: Ein ungerichteter zusammenhängender Graph $G = (V, E)$.
Gesucht ist das größte $k \in \mathbb{N}$, so dass G k-fach zusammenhängend ist.

$\Leftrightarrow$ Gesucht ist das größte $k \in \mathbb{N}$, so dass mindestens k Knoten aus G entfernt werden müssen, damit G unzusammenhängend wird.

$\Leftrightarrow$ gesucht ist die kleinste maximale Anzahl von knotendisjunkten Wegen zwischen zwei nicht adjazenten Knoten in G.

Ersetze jede ungerichtete Kante $\{u, v\}$ durch zwei gerichtete Kanten (u, v) und (v, u) und bestimme für jedes Paar s, t nicht adjazenter Knoten die maximale Anzahl von knotendisjunkten s, t-Wegen. Der kleinste gefundene Wert ist die Zusammenhangszahl.

Die Zusammenhangszahl kann in Zeit $O(|V|^2 \cdot |E| \cdot \sqrt{|V|})$ berechnet werden.

9 Methodische Grundlagen

9.1 Teile und Beherrsche

Auch bekannt als „divide and conquer“ oder „devide at impera“.

Strategie: Sei w eine Eingabe.

1. Teile die Eingabe w in $a \geq 2$ Teile $w_1, \ldots, w_a$ der Größe $\frac{|w|}{c}$ auf.
2. Löse die a Teilaufgaben mit den Eingaben $w_1, \ldots, w_a$.
3. Berechne die Lösung für w aus den Lösungen für $w_1, \ldots, w_a$.

Komplexität: (Aufteilen, Lösen der Teilaufgaben, Zusammenfügen)

Satz: (allgemeine Form) Sei $T : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion mit

1. $T(1) = b$ für ein $b \in \mathbb{N}$,
2. $T(n) = a \cdot T(\frac{n}{c}) + f(n)$ für $a, b, c \in \mathbb{N}$ und $f : \mathbb{N}_0 \to \mathbb{N}_0$.

Dann gilt:

$T(n) \in \Theta(n^{\log_c(a)})$	falls $f(n) \in O(n^{\log_c(a-\epsilon)})$ für ein $\epsilon > 0$
$T(n) \in \Theta(n^{\log_c(a)} \cdot \log(n))$	falls $f(n) \in \Theta(n^{\log_c(a)})$
$T(n) \in \Theta(f(n))$	falls $f(n) \in \Omega(n^{\log_c(a+\epsilon)})$ für ein $\epsilon > 0$ und $a \cdot f(n) \leq k \cdot f(n)$ für ein $k < 1$ und ein $n \geq n_0$

Ohne Beweis.

Satz: (vereinfachte Form) Sei $T : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion mit

1. $T(1) = b$ für ein $b \in \mathbb{N}$,
2. $T(n) = a \cdot T(\frac{n}{c}) + b \cdot n$ für $a, b, c \in \mathbb{N}$ und $n > 1$.

Dann gilt:

$$T(n) = \begin{cases} \Theta(n) & falls\ a < c \\ \Theta(n \cdot \log(n)) & falls\ a = c \\ \Theta(n^{\log_c(a)}) & falls\ a > c \end{cases}$$

Beweis: Sei ohne Beschränkung der Allgemeinheit (o.B.d.A.) $n = c^k$ für ein $k \in \mathbb{N}$

$$
\begin{aligned}
T(n) &= a \cdot T(\tfrac{n}{c}) + b \cdot n \\
&= a \cdot (a \cdot T(\tfrac{n}{c^2}) + \tfrac{b \cdot n}{c}) + b \cdot n \\
&= a \cdot (a \cdot (a \cdot T(\tfrac{n}{c^3}) + \tfrac{b \cdot n}{c^2}) + \tfrac{b \cdot n}{c}) + b \cdot n \\
&= a^k \cdot b + (\tfrac{a}{c})^{k-1} \cdot b \cdot n + (\tfrac{a}{c})^{k-2} \cdot b \cdot n + \ldots + b \cdot n \\
&= b \cdot n \cdot \textstyle\sum_{i=0}^{k} (\tfrac{a}{c})^i
\end{aligned}
$$

Falls $a < c$, dann konvergiert $\sum_{i=0}^{k} (\frac{a}{c})^i$ und somit ist $T(n) \in \Theta(n)$.
Falls $a = c$, dann ist $\sum_{i=0}^{k} (\frac{a}{c})^i = k+1 = \log_c(n)+1$ und somit $T(n) \in \Theta(n \cdot \log(n))$.
Falls $a > c$, dann gilt

$$
\begin{aligned}
b \cdot n \cdot \textstyle\sum_{i=0}^{k} (\tfrac{a}{c})^i &= b \cdot n \cdot \frac{(\frac{a}{c})^{1+k} - 1}{\frac{a}{c} - 1} \\
&= \frac{b \cdot n}{\frac{a}{c} - 1} \cdot (\frac{a^{1+k}}{c \cdot c^k} - 1) \\
&= \frac{b \cdot n}{\frac{a}{c} - 1} \cdot (\frac{a^{1+k}}{c \cdot n} - 1) \\
&= \frac{b}{(\frac{a}{c} - 1)c} \cdot a^{1+k} - \frac{b \cdot n}{\frac{a}{c} - 1} \\
&= \frac{b}{(\frac{a}{c} - 1)c} \cdot a \cdot a^{\log_c(n)} - \frac{b \cdot n}{\frac{a}{c} - 1} \\
&= \frac{b}{(\frac{a}{c} - 1)c} \cdot a \cdot n^{\log_c(a)} - \frac{b \cdot n}{\frac{a}{c} - 1} \quad \in \Theta(n^{\log_c(a)})
\end{aligned}
$$

(Bemerkung: $a^{\log_c(n)} = n^{\log_c(a)}$)

Beispiel: Sortieren durch Mischen

Eingabe: $x_1, \ldots, x_n \in \mathbb{N}$ mit $n = 2^k$
Ausgabe: $x_{\pi(1)}, \ldots, x_{\pi(n)}$ wobei π eine Permutation der Zahlen $1, \ldots, n$ ist mit der Eigenschaft, dass $x_{\pi(i)} \leq x_{\pi(i+1)}$ für $1 \leq i < n$.

Algorithmus:

Teile die Eingabe in zwei gleich große Teilfolgen T_1, T_2 auf, sortiere beide Teilfolgen T_1, T_2 mit dem gleichen Algorithmus und mische die entstehenden sortierten Teilfolgen anschließend zu einer sortierten Gesamtfolge.

$a = 2$, $c = 2$, $b = 1 \Rightarrow$ Laufzeit: $O(n \cdot \log(n))$

Beispiel: Maximumsuche

Eingabe: $x_1, \ldots, x_n \in \mathbb{N}$ mit $n = 2^k$
Ausgabe: $\max\{x_1, \ldots, x_n\}$

Algorithmus:

Vergleiche benachbarte Elemente und bilde so aus den jeweils größeren Elementen eine neue Teilfolge mit $\frac{n}{2}$ Elemente. Suche das Maximum in der neuen Teilfolge.

$a = 1, c = 2, b = 1 \Rightarrow$ Laufzeit: $O(n)$

Beispiel: Matrixmultiplikation

Eingabe: Zwei $N \times N$ Matrizen $A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$ und $B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$ mit $N = 2^k$ $(n = 2^{2k+1})$

Ausgabe: $C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} = A \times B$

Die Schulmethode zur Multiplikation zweier $N \times N$ Matrizen hat eine Laufzeit von $O(N^3)$.

Einer der schnellsten Algorithmen hat eine Laufzeit von $O(N^{2,376\ldots})$.

Eine untere Schranke ist $O(N^2)$.

Beispiel für 2×2-Matrizen:

$C = A \cdot B$
$c_{1,1} = a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1}$
$c_{1,2} = a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{2,2}$
$c_{2,1} = a_{2,1} \cdot b_{1,1} + a_{2,2} \cdot b_{2,1}$
$c_{2,2} = a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2}$

$\Longrightarrow$ 8 Multiplikationen und 4 Additionen

Verbesserung: Straßen (1969)

$m_1 = (a_{1,2} - a_{2,2}) \cdot (b_{2,1} + b_{2,2})$
$m_2 = (a_{1,1} + a_{2,2}) \cdot (b_{1,1} + b_{2,2})$
$m_3 = (a_{2,1} - a_{1,1}) \cdot (b_{1,1} + b_{1,2})$
$m_4 = (a_{1,1} + a_{1,2}) \cdot b_{2,2}$
$m_5 = a_{1,1} \cdot (b_{1,2} - b_{2,2})$
$m_6 = a_{2,2} \cdot (b_{2,1} - b_{1,1})$
$m_7 = (a_{2,1} + a_{2,2}) \cdot b_{1,1}$

$c_{1,1} = m_1 + m_2 - m_4 + m_6$
$c_{1,2} = m_4 + m_5$

$c_{2,1} = m_6 + m_7$
$c_{2,2} = m_2 + m_3 + m_5 - m_7$

$\Longrightarrow$ 7 Multiplikationen und 18 Additionen

Multipliziere größere Matrizen mittels Teile und Beherrsche.

Sei n eine Zweierpotenz.
$T(N) = a \cdot T(\frac{N}{c}) + b \cdot N$
$a = 7, c = 2, b = \frac{18}{2}$

(7 Multiplikationen von $\frac{N}{2} \times \frac{N}{2}$-Matrizen und 18 Additionen von $\frac{N}{2} \times \frac{N}{2}$-Matrizen)

$a > c \Rightarrow T(n) = O(N^{\log_c(a)}) = O(N^{\log_2(7)}) = O(N^{2,81\ldots})$

Bemerkung:

1979:	Pan	$O(N^{2,522\ldots})$
1979:	Schönhage	$O(N^{2,508\ldots})$
1982:	Coppersmith / Winograd	$O(N^{2,4156\ldots})$
1986:	Straßen	$O(N^{2,41\ldots})$
1987:	Coppersmith / Winograd	$O(N^{2,376\ldots})$
i?		

Beispiel: MinMax-Suche (aufgepasst!)

Eingabe: $x_1, \ldots, x_n \in \mathbb{N}$ mit $n = 2^k$
Ausgabe: $\min\{x_1, \ldots, x_n\}$ und $\max\{x_1, \ldots, x_n\}$

Algorithmus:

Teile die Eingabe in zwei gleich große Teilfolgen T_1, T_2 auf, bestimme
$\min(T_1), \max(T_1)$ und $\min(T_2), \max(T_2)$ und berechne daraus $\min(T)$ und $\max(T)$.

Komplexitätsmaß = Anzahl der Vergleiche

$a = 2$, $c = 2$, $b =$?

Sei $T(n) = 1$ falls $n = 2$

$$\begin{aligned} T(n) &= 2 \cdot T(\tfrac{n}{2}) + 2 \text{ falls } n > 2 \\ &= \underbrace{2 \cdot (2 \cdot (2 \cdot (\ldots \cdot (2}_{k-1} + 2) \ldots + 2) + 2) + 2 \\ &= 2^{k-1} + 2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2 \\ &= 2^{k-1} + 2^k - 2 \\ &= \tfrac{n}{2} + n - 2 \qquad \in O(n) \end{aligned}$$

9.2 Rekursive Algorithmen

Im Gegensatz zu „Teile und Beherrsche“ findet bei rekursiven Algorithmen nicht notwendigerweise eine Zerlegung in gleiche Teilprobleme statt!

Beispiel: Fibonacci-Zahlen

$Fib(n) = Fib(n-1) + Fib(n-2)$ falls $n > 2$
$Fib(n) = 1$ falls $1 \leq n \leq 2$.

Laufzeit: $O(1.618^n)$

Rekursive Algorithmen haben meistens eine schlechte (exponentielle) Laufzeit, da interessante Zwischenergebnisse oft mehrfach berechnet werden.

9.3 Dynamische Programmierung

Grundidee: Systematische einmalige Berechnung aller notwendigen Zwischenergebnisse

Beispiel: Fibonacci-Zahlen

$Fib(1) = 1, Fib(2) = 1, Fib(3) = 2, Fib(4) = 3, Fib(5) = 5, Fib(6) = 8, \ldots$

Aus den bisher berechneten Teillösungen werden zum Weiterrechnen nur die letzten zwei Werte benötigt.

Laufzeit: $O(n)$

Beispiel: Minimale Triangulation von konvexen Polygonen

- Seien
$$p_1 = (x_1, y_1),\ p_2 = (x_2, y_2) \in \mathbb{R}^2$$
zwei Punkte im 2-dimensionalen euklidischen Raum $\mathbb{R}^2$.
- Das Liniensegment zwischen p_1 und p_2 ist
$$\overline{p_1 p_2} := \{\alpha \cdot p_1 + (1-\alpha) \cdot p_2 \mid \alpha \in \mathbb{R},\ 0 \leq \alpha \leq 1\}.$$
- Die euklidische Distanz zwischen p_1 und p_2 ist
$$d(p_1, p_2) := \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$
- Eine Menge $D \subseteq \mathbb{R}^2$ heißt konvex, falls für alle $p_i, p_j \in D$: $\overline{p_i p_j} \subseteq D$.

Gegeben: Ein konvexes Polygon $P = (A_1, \ldots, A_n)$, $n > 3$, $A_i \in \mathbb{R}^2$, $1 \leq i \leq n$, im Euklidischen Raum $\mathbb{R}^2$

Gesucht ist eine Teilmenge

$$T \subseteq \{(A_i, A_j) \mid 1 \leq i, j \leq n,\ |i - j| > 1\}$$

mit minimalen Kosten

$$C(T) = \sum_{(A_i, A_j) \in T} d(A_i, A_j),$$

die das Polygon trianguliert, d.h., $|T| = n - 3$ und keine 2 Kanten aus T kreuzen sich.

Algorithmus zur Berechnung der Kosten einer minimalen Triangulation für konvexe Polygone:

(1) for $i = 1, \ldots, n-1$ {
(2) $\quad c_{i,i+1} := -d(p_i, p_{i+1})$; }
(3) for $i = 1, \ldots, n-2$ {
(4) $\quad c_{i,i+2} := 0$; }
(5) for $s = 3, \ldots, n-1$ {
(6) $\quad$ for $i = 1, \ldots, n-s$ {
(7) $\quad\quad c_{i,i+s} := \infty$;
(8) $\quad\quad$ for $k = i+1, \ldots, i+s-1$ {
(9) $\quad\quad\quad c_{i,i+s} := min\{c_{i,i+s}, c_{i,k} + c_{k,i+s} + d(p_i, p_k) + d(p_k, p_{i+s})\}$; } } }

Für $j - i > 2$ sind $c_{i,j}$ die Kosten einer minimalen Triangulation des Polygons.

Ausgabe: $c_{1,n}$

Laufzeit: $O(n^3)$

9.4 Lokale Suche

Sei P ein algorithmisches Problem.
Sei I die Menge der Instanzen.
Sei S die Menge der Lösungen.
Sei $f : S \longrightarrow \mathbb{R}^+$ eine Kostenfunktion für die Lösungen.

Gegeben: $w \in I$

Gesucht: $s \in S$ mit minimalen Kosten (s ist Lösung für w)

Die lokale Suche benötigt einfach zu berechnende Nachbarschaftsbeziehungen zwischen den Lösungen.

Suche von einer gegebenen Lösung $s \in S$ eine bessere Lösung s' aus der näheren Umgebung von s. Fahre mit s' fort.

Beispiel: Traveling Salseman Problem (TSP)

Gegeben: Eine $N \times N$ Kostenmatrix $(c_{i,j})_{c_{i,j} \in \mathbb{R}}$

Gesucht: Eine Permutation π der Zahlen $1, \ldots, N$, so dass die Kosten der Permutation

$$C(\pi) = c_{\pi(N),\pi(1)} + \sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)}$$

minimal sind.

Algorithmus:

1. Starte mit einer beliebigen Permutation π der Zahlen $1, \ldots, N$ und wiederhole so lange wie möglich die nachfolgende Anweisung:
2. Wähle zwei beliebige Zahlen $i, j \in \{1, \ldots, N\}$ und vertausche $\pi(i)$ mit $\pi(j)$, falls die Kosten der neuen Permutation geringer sind als die Kosten der alten Permutation.

Bemerkung: Die lokale Suche liefert für das TSP nicht immer eine optimale Lösung.

9.5 Greedy-Strategie

Greedy bedeutet, dass eine Lösung bzw. Teillösung in jedem Schritt um den maximal möglichen Wert verbessert wird.

Beispiel: TSP

Algorithmus:

1. Starte mit einer beliebigen Permutation π der Zahlen $1, \ldots, N$ und wiederhole so lange wie möglich die nachfolgende Anweisung:

2. Wähle zwei Zahlen $i, j \in \{1, \ldots, N\}$ für die die neue Permutation, die man durch Vertauschen von $\pi(i)$ und $\pi(j)$ erhält, die geringsten Kosten hat. Vertausche $\pi(i)$ mit $\pi(j)$.

Bemerkung: Die Greedy-Strategie liefert ebenfalls für das TSP nicht immer eine optimale Lösung.

Beispiel: Bezahlen mit Geldscheinen/-münzen

Gegeben: Geldbetrag: 137,22 Euro

Aufgabe: Bezahle den Betrag mit möglichst wenigen Scheinen/Münzen

137,22 = 100 + 20 + 10 + 5 + 2 + 0,20 + 0,02

Funktioniert mit Greedy-Algorithmen mit der Aufteilung 10, 20, 50, 100, ..., aber nicht mit allen Aufteilungen, z.B. nicht mit der Aufteilung 10, 20, 40, 50, 100, ...

Greedy Lösung: 80 = 50 + 20 + 10
Optimale Lösung: 80 = 40 + 40

10 Geometrische Algorithmen

Betrachte Punkte und Punktmengen im Euklidischen Raum $\mathbb{R}^d$.

- $p = (x_1, \ldots, x_d) \in \mathbb{R}^d$ ist ein Punkt
- für $p_1, p_2 \in \mathbb{R}^d$ ist $\{\alpha \cdot p_1 + (1-\alpha) \cdot p_2 \mid 0 \leq \alpha \leq 1\}$ ein Liniensegment, andere Schreibweise: $\overline{p_1 p_2}$
- eine Menge $D \subseteq \mathbb{R}^d$ heißt konvex, falls für alle $p_1, p_2 \in D$ gilt: $\overline{p_1 p_2} \subseteq D$

10.1 Konvexe Hülle

Gegeben: eine Menge $P = \{p_1, \ldots, p_n\}$ von Punkten im zweidimensionalen Raum.

Gesucht: die kleinste konvexe Menge, die P enthält (also ein konvexes Polygon).

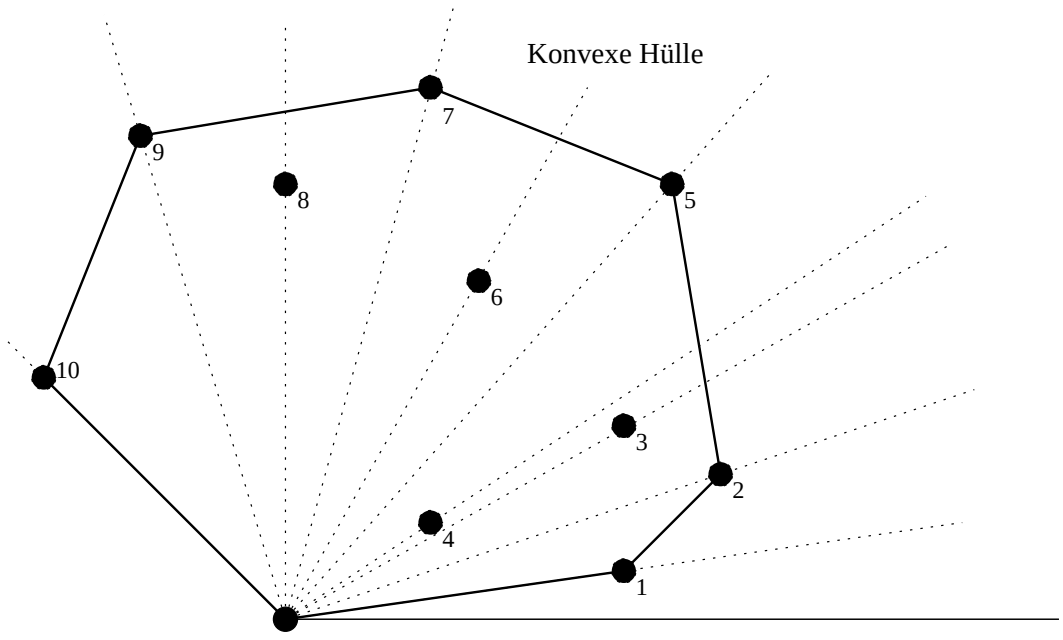


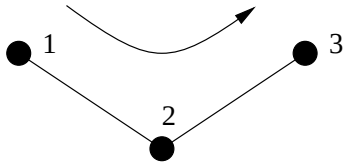
Abbildung 10.1: Konvexe Hülle einer Punktmenge P

Graham-Scan

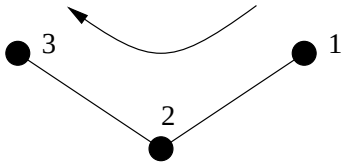
- Suche einen Punkt p mit kleinster y-Koordinate (falls mehrere existieren, wähle den mit kleinster x-Koordinate)
- Sortiere alle Punkte aufsteigend nach Winkeln zu einer gedachten Horizontalen mit Anfangspunkt p. Die Punkte werden in einer doppeltverketteten Liste mit Zeigern 'next' und 'pred' gespeichert.
- (1) $v := p$
  (2) while (next(v) $\neq p$) {

(3) if ((v,next(v),next(next(v))) ist 'Linksdrehung') {
(4) v := next(v) }
(5) else {
(6) delete next(v)
(7) v := pred(v) }
(8) }

Bemerkung
Wird eine Rechtsdrehung festgestellt, so wird v auf pred(v) zurückgesetzt, da v nicht unbedingt auf der konvexen Hülle liegt (siehe z.B. Knoten 3 in Abbildung 10.1).



Linksdrehung (ccw / counter clock wise)



Rechtsdrehung (cw / clock wise)

gegeben: 3 Punkte $p_0 = (x_0, y_0), p_1 = (x_1, y_1), p_2 = (x_2, y_2)$
$\Delta x_1 = x_1 - x_0$
$\Delta y_1 = y_1 - y_0$
$\Delta x_2 = x_2 - x_0$
$\Delta y_2 = y_2 - y_0$
$\Delta y_1 \cdot \Delta x_2 \leq \Delta y_2 \cdot \Delta x_1$ ist Linksdrehung
$\Delta y_1 \cdot \Delta x_2 > \Delta y_2 \cdot \Delta x_1$ ist Rechtsdrehung

Laufzeit: $O(n \cdot log(n))$

Platz: $O(n)$

10.2 Das Scan-Line Prinzip

Geeignet für zweidimensionale geometrische Probleme.

Idee: Lasse eine Scan-Line (vertikale Linie) von links nach rechts über eine gegebene Menge von Objekten in der Ebene laufen. Damit zerlegt man ein zweidimensionales Problem in eine Folge von eindimensionalen Problemen.
Während man die Scan-Line über die Eingabemenge schwenkt, hält man eine Vertikalstruktur L aufrecht, in der man sich alle für das jeweilige Problem benötigten Daten merkt.

Allgemeines Liniensegment-Schnittproblem

Gegeben: eine Menge von Punktpaaren im zweidimensionalen Raum
$S = \{s_1, \ldots, s_n\}$, wobei $s_i = \{(x_i, y_i), (x_i', y_i') \mid x_i, y_i, x_i', y_i' \in \mathbb{R}\}$.
Gesucht: Schnittpunkte der Liniensegmente in S.

Schnittpunkttest: Teste, ob es zwei Liniensegmente in S gibt, die sich schneiden.

Schnittpunktaufzählung: Zähle alle Schnittpunkte der Segmente in S auf.

Zur Vermeidung von Sonderfällen sei vorausgesetzt:

- Es gibt keine vertikalen Liniensegmente.
- In jedem Punkt schneiden sich höchstens zwei Segmente.
- Alle Anfangs- und Endpunkte haben paarweise verschiedene x-Koordinaten.

Definition:
Seien A und B zwei Liniensegmente. A heißt x-oberhalb von B (in Zeichen: $A \uparrow_x B$), wenn die vertikale Gerade durch x sowohl A als auch B schneidet, und der Schnittpunkt der Geraden mit A oberhalb des Schnittpunkts der Geraden mit B liegt.

Beispiel:

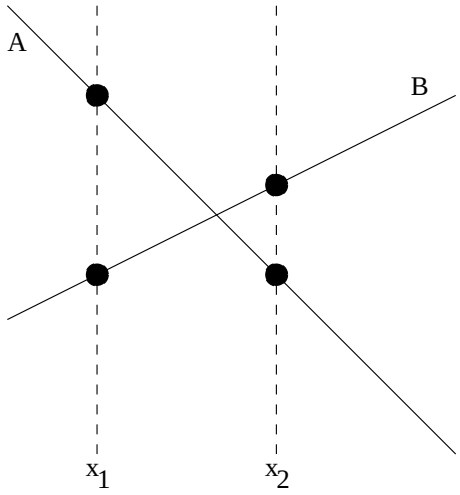


Abbildung 10.2: $A \uparrow_{x_1} B$ und $B \uparrow_{x_2} A$

Beobachtung:
Wenn sich zwei Segmente A und B schneiden, dann gibt es eine Stelle x links von dem Schnittpunkt, so daßA und B in der Ordnung unmittelbar aufeinanderfolgen (da sich in einem Punkt höchstens zwei Segmente schneiden).

Überprüfe also für je zwei Segmente, ob sie sich schneiden, sobald sie an einer Stelle x bzgl. $\uparrow_x$ unmittelbar benachbart sind, und kein Schnittpunkt kann übersehen werden.

Algorithmus Schnittpunkttest

Q = Folge der $2 \cdot n$ Anfangs- und Endpunkte der Liniensegmente
sortiert nach aufsteigenden x-Koordinaten
$L = \emptyset$
gefunden = false
while (Q ist nicht leer) und (not gefunden)
{ (x, y) = nächster Punkt aus Q
if ((x, y) ist linker Endpunkt eines Segments s) then
{ füge s entsprechend der Ordnung $\uparrow_x$ in L ein
bestimme den Nachfolger s' von s in L bzgl. $\uparrow_x$
bestimme den Vorgänger s'' von s in L bzgl. $\uparrow_x$
if $(s \cap s' \neq \emptyset)$ oder $(s \cap s'' \neq \emptyset)$ then gefunden = true
}
else
{ (* (x, y) ist rechter Endpunkt eines Segments s *)

bestimme den Nachfolger s' von s in L bzgl. $\uparrow_x$
bestimme den Vorgänger s'' von s in L bzgl. $\uparrow_x$
entferne s aus L
if $(s' \cap s'' \neq \emptyset)$ then gefunden = true
}
}

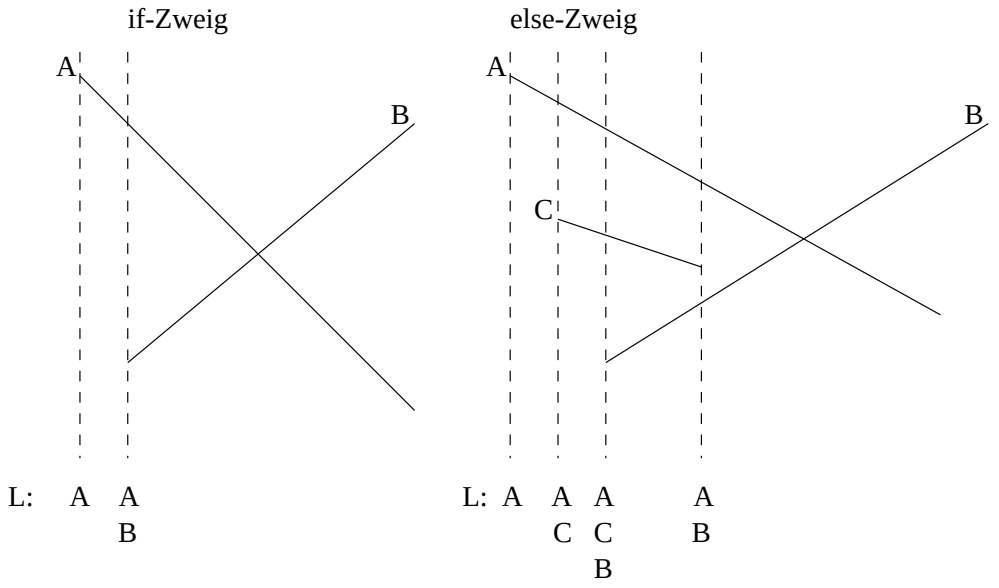


Abbildung 10.3: Schnittpunkttest

Man implementiert L als balancierten Suchbaum.

Die Operationen Entfernen, Einfügen, Bestimmen des Vorgängers und Nachfolgers können in Zeit $O(n \cdot log(n))$ ausgefhrt werden.

Laufzeit: $O(n \cdot log(n))$

Platz: $O(n)$

Modifiziere den Algorithmus so, daß das Verfahren fortgesetzt wird, wenn ein Schnittpunkt gefunden wurde. Die lokale Ordnung der Vertikalstruktur L muß dabei korrekt bleiben, d.h. immer wenn die Scan-Line einen Schnittpunkt zweier Segmente A, B passiert, wechseln A und B ihren Platz in L. Daher muß die Scan-Line auch an allen gefundenen Schnittpunkten angehalten werden; jeder Schnittpunkt muß in Q eingefügt werden.

Um Sonderfälle zu vermeiden, wird vorausgesetzt, daß die x-Koordinaten der Schnittpunkte paarweise verschieden sind und nicht mit Anfangs- oder Endpunkten von Liniensegmenten zusammenfallen.

Algorithmus Schnittpunktaufzählung:

Q = Folge der $2 \cdot n$ Anfangs- und Endpunkte der Liniensegmente
sortiert nach aufsteigenden x-Koordinaten
$L = \emptyset$
while (Q ist nicht leer)
{ (x, y) = nächster Punkt aus Q
if ((x, y) ist linker Endpunkt eines Liniensegments s) then
{ füge s entsprechend der Ordnung $\uparrow_x$ in L ein
bestimme den Nachfolger s' von s in L bzgl. $\uparrow_x$
bestimme den Vorgänger s'' von s in L bzgl. $\uparrow_x$
if ($s \cap s' \neq \emptyset$) then füge den Schnittpunkt von s und s' in Q ein
if ($s \cap s'' \neq \emptyset$) then füge den Schnittpunkt von s und s'' in Q ein
}
else if ((x, y) ist ein rechter Endpunkt eines Segments s) then
{ bestimme den Nachfolger s' von s in L bzgl. $\uparrow_x$
bestimme den Vorgänger s'' von s in L bzgl. $\uparrow_x$
entferne s aus L
if ($s' \cap s'' \neq \emptyset$) then füge den Schnittpunkt von s' und s'' in Q ein
}
else
{ (* (x, y) ist Schnittpunkt zweier Segmente s' und s''; s' ist oberhalb von s'' in L *)
gib den Schnittpunkt (x, y) aus
vertausche s' und s'' in L (* $\Rightarrow$ s'' oberhalb von s' *)
bestimme den Nachfolger t' von s' in L bzgl. $\uparrow_x$
bestimme den Vorgänger t'' von s'' in L bzgl. $\uparrow_x$
if ($s' \cap t' \neq \emptyset$) then füge den Schnittpunkt von s' und t' in Q ein
if ($s'' \cap t'' \neq \emptyset$) then füge den Schnittpunkt von s'' und t'' in Q ein
}
}

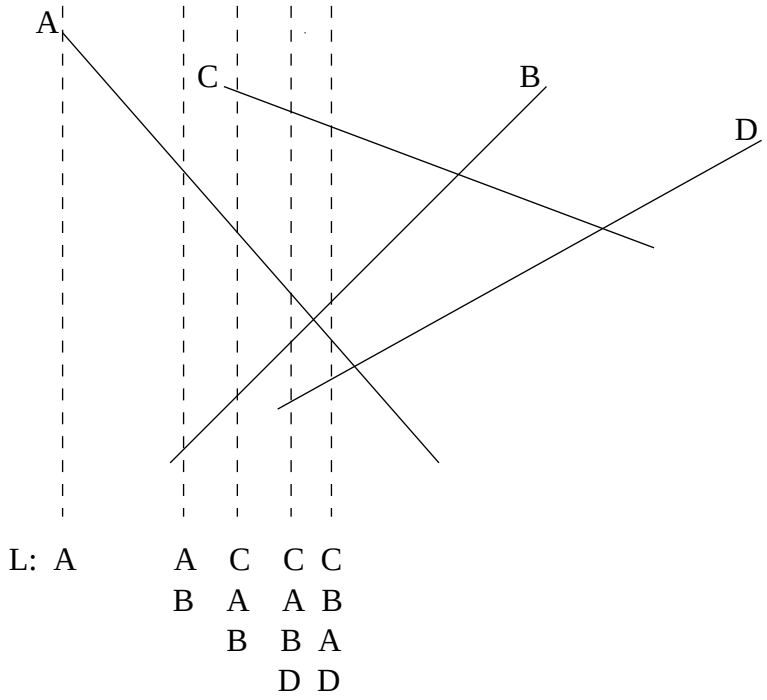


Abbildung 10.4: Schnittpunktaufzählung

Implementierung von Q:
Man organisiert Q als balancierten Binärbaum (z.B. AVL-Baum, d.h. Höhe des linken Teilbaums = Höhe des rechten Teilbaums $\pm$ 1), damit die Operationen Suchen, Einfügen, Entfernen gut unterstützt werden.

Laufzeit: Die Schleife wird für k Schnittpunkte genau $(2 \cdot n + k)$-mal durchlaufen
$\Rightarrow O((n + k) \cdot log(n))$

Platz: Anzahl der Schnittpunkte + Anzahl der Anfangs-/Endpunkte
$\Rightarrow O(n + k)$

10.3 Geometrisches Divide and Conquer

Schnittproblem für iso-orientierte Liniensegmente:

Gegeben: eine Menge $S = \{s_1, \ldots, s_n\}$ vertikaler und horizontaler Liniensegmente im zweidimensionalen Raum.

Gesucht: alle Schnittpunkte der Segmente in S

Da jedes Segment durch das Paar seiner Endpunkte repräsentiert wird, läßt sich die Menge S einfach und eindeutig durch eine vertikale Gerade in eine linke und rechte Hälfte teilen.

Beispiel:

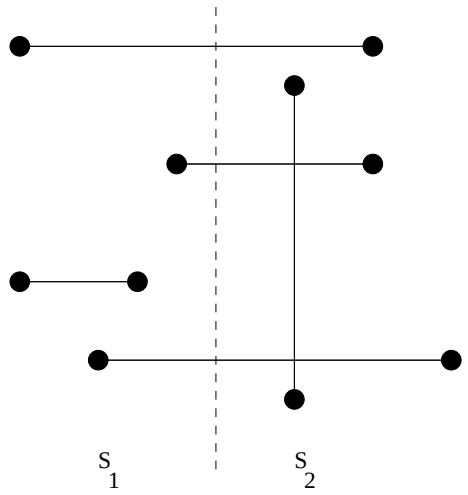


Abbildung 10.5: Teilung der Segmente

Algorithmus ReportCuts(S):

divide:	Falls S mehr als ein Element enthält, teile S durch eine vertikale Gerade in eine linke Hälfte S_1 und eine rechte Hälfte S_2 auf.
conquer:	ReportCuts(S_1) ReportCuts(S_2) $\Rightarrow$ Ausgabe aller Schnitte in S_1 und S_2
merge:	a) Bericht aller Schnitte zwischen vertikalen Segmenten in S_1 und horizontalen Segmenten mit rechtem Endpunkt in S_2, deren linker Endpunkt nicht in S_1 oder S_2 liegt. b) Bericht aller Schnitte zwischen vertikalen Segmenten in S_2 und horizontalen Segmenten mit linkem Endpunkt in S_1, deren rechter Endpunkt nicht in S_1 oder S_2 liegt.

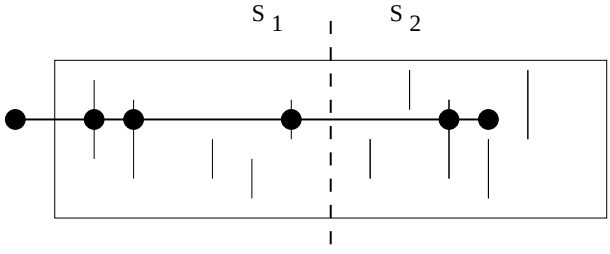


Abbildung 10.6: zu a)

Rekursionsinvariante:
Nach ReportCuts(S) sind alle Schnitte zwischen vertikalen Segmenten aus S und horizontalen Segmenten, deren linker oder rechter Endpunkt (oder beide) in S liegt, ausgegeben worden.

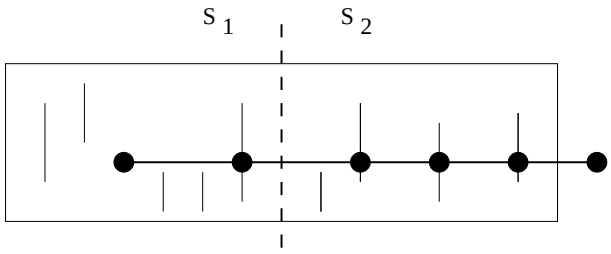
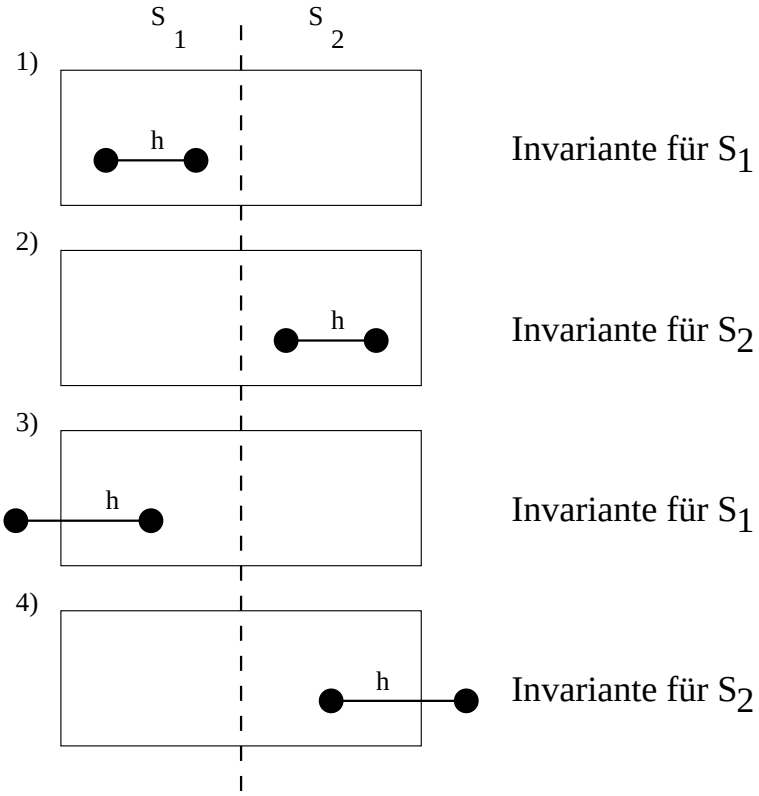


Abbildung 10.7: zu b)

Beweis:

Induktionsanfang:
Wenn S nur aus einem Element besteht, gilt die Aussage trivialerweise; es werden keine Schnitte ausgegeben und das Verfahren bricht ab.

Induktionsschritt:
Sei h ein horizontales Segment, dessen linker oder rechter Endpunkt (oder beide) in S liegt.



Algorithmische Umsetzung:

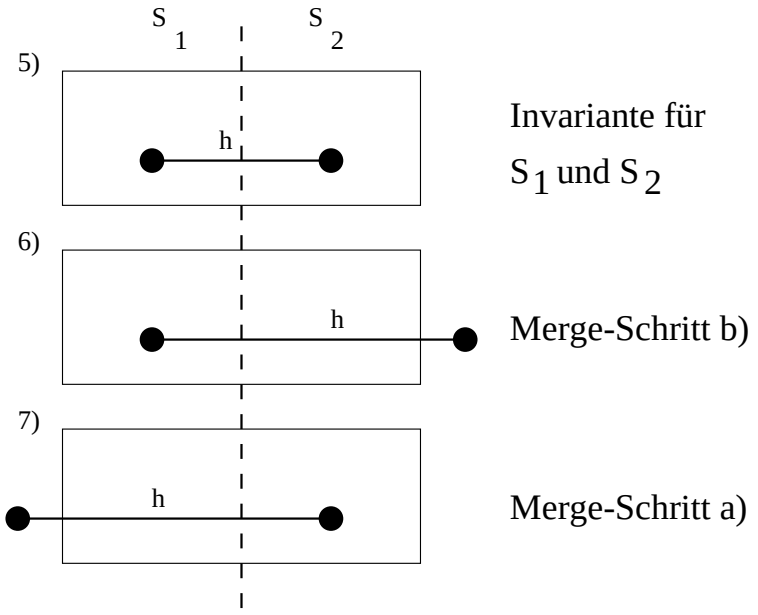


Abbildung 10.8: Möglichkeiten für h beim Induktionsschritt

$L(S)$: Menge der y-Koordinaten horizontaler Segmente, deren linker Endpunkt in S, deren rechter Endpunkt aber nicht in S liegt

$R(S)$: Menge der y-Koordinaten horizontaler Segmente, deren rechter Endpunkt in S, deren linker Endpunkt aber nicht in S liegt

$V(S)$: Menge der durch die vertikalen Segmente in S definierten y-Intervalle

Bezeichne $y(h)$ die y-Koordinate eines horizontalen Segments h und bezeichne $y_u(v)$ bzw. $y_o(v)$ die untere bzw. obere y-Koordinate eines vertikalen Segments v.

Merge: Bestimme alle Paare (h, v)

a) $y(h) \in R(S_2) - L(S_1)$
$[y_u(v), y_o(v)] \in V(S_1)$
$y_u(v) \leq y(h) \leq y_o(v)$

b) $y(h) \in L(S_1) - R(S_2)$
$[y_u(v), y_o(v)] \in V(S_2)$
$y_u(v) \leq y(h) \leq y_o(v)$

$\Rightarrow$ Die $S = S_1 \cup S_2$ zugeordneten Mengen bestimmen sich wie folgt:
$L(S) = (L(S_1) - R(S_2)) \cup L(S_2)$
$R(S) = (R(S_2) - L(S_1)) \cup R(S_1)$

$V(S) = V(S_1) \cup V(S_2)$

Datenstrukturen:

S:	nach aufsteigenden x-Koordinaten sortiertes Array
$L(S), R(S)$:	nach aufsteigenden y-Werten sortierte verkettete Listen
$V(S)$:	nach y_u-Werten sortierte, verkettete Liste

Laufzeit: $O(1)$ (divide) + $2 \cdot T(\frac{n}{2})$ (conquer) + $O(n)$ (merge)
$\Rightarrow O(n \cdot log(n))$
bei k Paaren sich schneidender Linienelemente $\Rightarrow O(n \cdot log(n) + k)$

Platz: $O(n)$

10.4 Distanzprobleme und Voronoi-Diagramme

Wir beschränken uns auf die Euklidische Ebene $\mathbb{R}^2$ mit der Distanzfunktion $d(p_1, p_2)$,
$p_1 = (x_1, y_1), p_2 = (x_2, y_2)$, $d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Dichtestes Punktpaar:

Gegeben: Menge P von n Punkten in der Ebene
Gesucht: Paar $p_1, p_2 \in P$ mit minimaler Distanz $d(p_1, p_2)$

Lösbar in $O(n^2)$ Schritten, benötigt aber mindestens $\Omega(n \cdot log(n))$ Schritte!

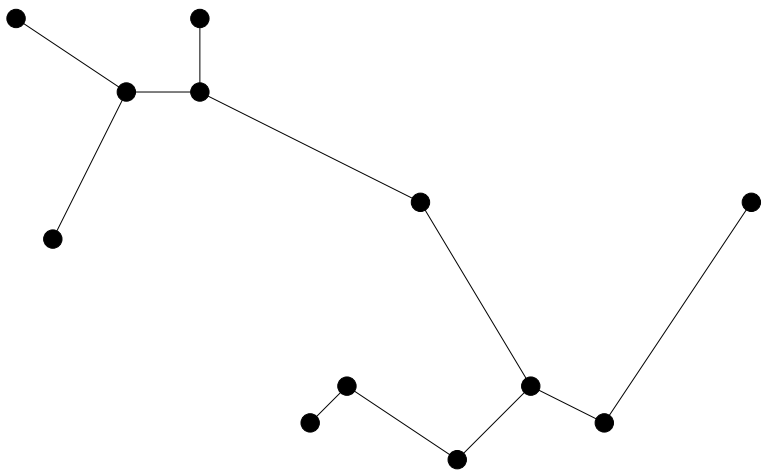


Abbildung 10.9: Beispiel für Nachbarschaftsprobleme

Das Problem, für eine gegebene Folge von n Zahlen festzustellen, ob eine Zahl mehrmals in der Folge auftritt (element uniqueness), benötigt zur Lösung $\Omega(n \cdot log(n))$ Schritte (F.P. Preparata).

Alle nächsten Nachbarn:

Gegeben: Menge P von n Punkten in der Ebene
Gesucht: Für jeden Punkt $p_1 \in P$ ein nächster Nachbar $p_2 \in P$, d.h. ein Punkt $p_2 \neq p_1$ mit $d(p_1, p_2) = min_{p \in P - \{p_1\}}\{d(p_1, p)\}$

Lösbar in Zeit $O(n^2)$, benötigt mindestens $\Omega(n \cdot log(n))$ Schritte

Minimaler spannender Baum:

Gegeben: Menge P von n Punkten in der Ebene
Gesucht: Ein minimaler spannender Baum für P, d.h. ein Baum, dessen Knoten die Punkte aus P und dessen Kanten Verbindungen zwischen Punkten aus P sind und der unter allen solchen Bäumen minimale Länge hat. (Die Länge ist die Summe der Längen der Kanten.)

Benötigt mindestens $\Omega(n \cdot log(n))$ Schritte.

Suche nächsten Nachbarn:

Gegeben: Menge P von n Punkten in der Ebene
Gesucht: Datenstruktur und Algorithmen, die

- P in der durch die Datenstruktur vorgegebenen Form speichert (preprocessing)
- zu einem gegebenen neuen Punkt q (query point) einen Punkt aus P findet, der nächster Nachbar von q ist.

Allgemein kann jede Anfrage in $O(n)$ Schritten durchgeführt werden.

Das Voronoi-Diagramm:
Sei $H(p_1|p_2)$ die Menge der Punkte, die näher bei p_1 als bei p_2 liegen. Für einen Punkt $p_1 \in P$ ist die Menge aller Punkte p, die näher bei p_1 als bei irgendeinem anderen Punkt aus P liegen, die Voronoi-Region $VR(p_1)$ für p_1.
Die Menge aller Voronoi-Regionen ist das Voronoi-Diagramm $VD(P)$.
Betrachte das Voronoi-Diagramm als ein Netzwerk. Die Knoten heißen Voronoi-Knoten, die Kanten heißen Voronoi-Kanten.

Bemerkungen:

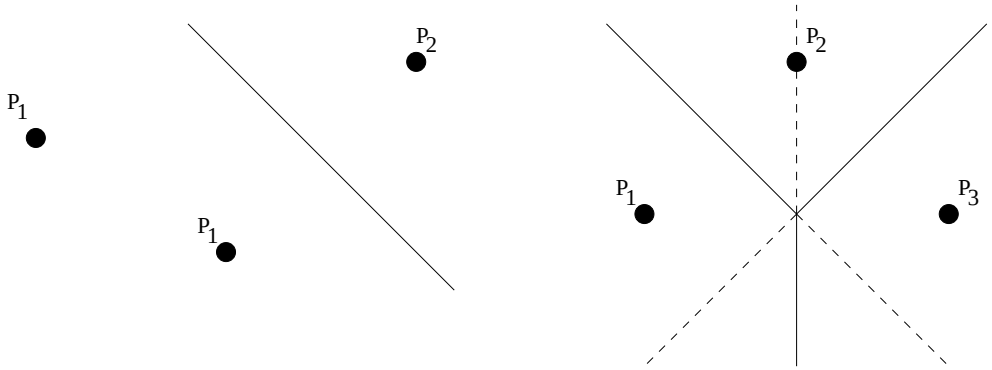


Abbildung 10.10: Beispiele für Voronoi-Diagramme

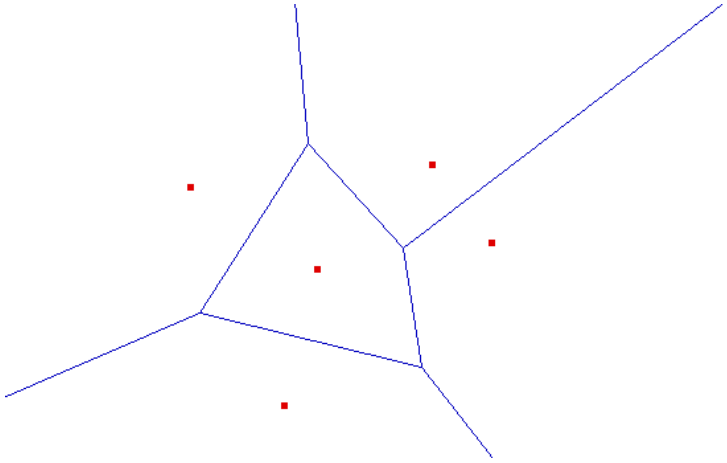


Abbildung 10.11: Die Voronoi-Region $VR(p_1)$ für Punkt p_1

- Wenn keine 4 Punkte auf einem gemeinsamen Kreis liegen, dann hat jeder Voronoi-Knoten den Grad 3.
- Nächste Nachbarn haben sich berührende Voronoi-Regionen.

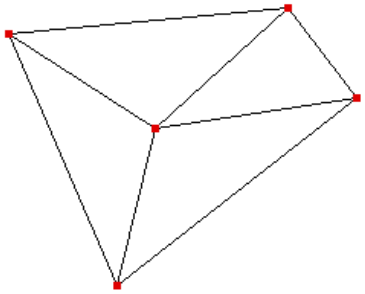


Abbildung 10.12: Die Delaunay-Triangulierung für diese Punktmenge

- Die Voronoi-Regionen der Punkte auf der konvexen Hülle sind unbeschränkt (und umgekehrt).

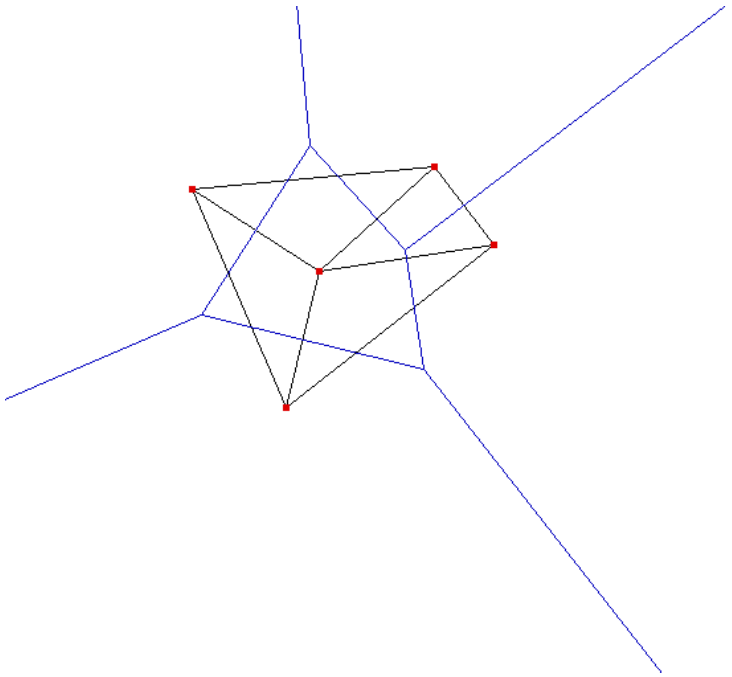


Abbildung 10.13: Voronoi-Diagramm und Delaunay-Triangulierung

Delaunay-Triangulierung:
Zwei Punkte sind verbunden, wenn sich ihre Voronoi-Regionen berühren.
Jedes Voronoi-Diagramm hat höchstens $2 \cdot n - 4$ Knoten und höchstens $3 \cdot n - 6$ Kanten (folgt aus den Eulergleichungen für planare Graphen).

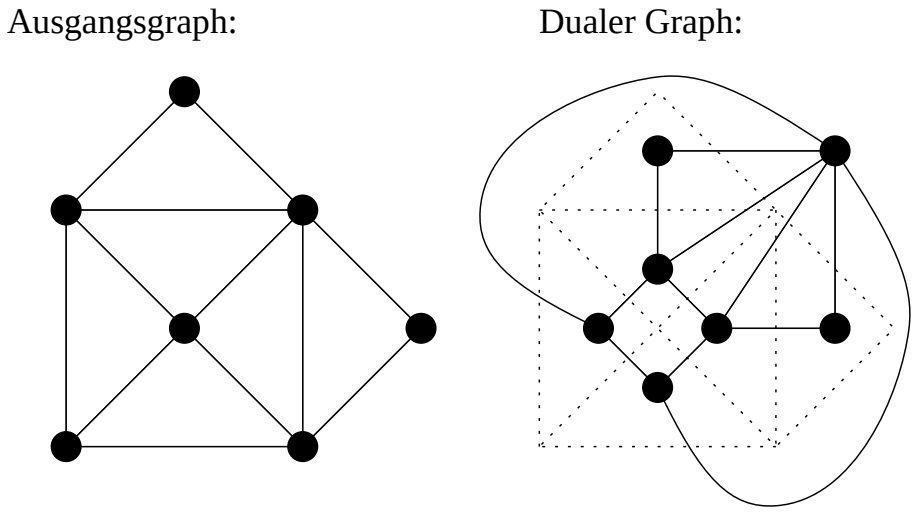


Abbildung 10.14: Beispiel für einen dualen Graphen

Speicherung des Voronoi-Diagramms:

Man speichert die Voronoi-Regionen in einer doppelt verketteten Liste der Kanten.

Jede Kante hat einen Verweis auf den Anfangs- und den Endknoten, die linke und die rechte Fläche, die nächste Kante in der linken und die nächste in der rechten Fläche (im Uhrzeigersinn).

Die Richtung der Kante ist implizit willkürlich durch Abspeicherung festgelegt.

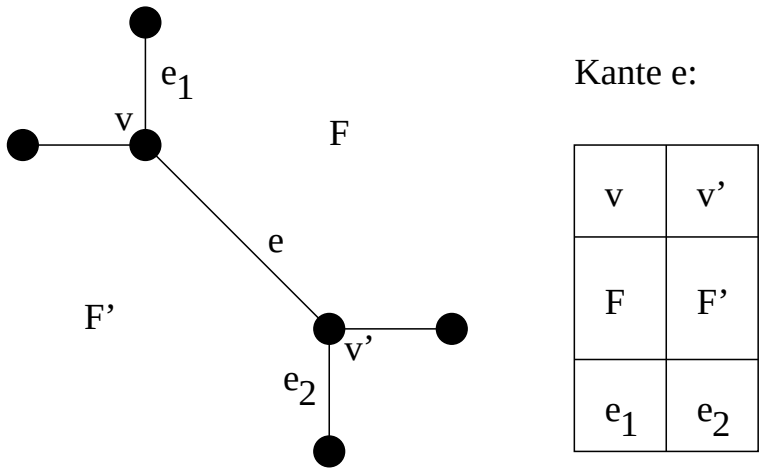


Abbildung 10.15: Möglichkeit, eine Kante zu speichern

Konstruktion des Voronoi-Diagramms:

Problem: Voronoi-Diagramm
Gegeben: Menge von Punkten P in der Ebene
Gesucht: Voronoi-Diagramm für P als doppelt verkettete Kantenliste

Teile und Beherrsche:

- Teile P in etwa gleich große Teilmengen P_1 und P_2 auf.
- Berechne die Voronoi-Diagramme für P_1 und P_2 rekursiv.
- Verschmelze die Voronoi-Diagramme $VD(P_1)$ und $VD(P_2)$ zu $VD(P)$.

Das Voronoi-Diagramm für einen Punkt ist die gesamte Ebene.

Beobachtung:

Wenn die Menge P durch eine vertikale Linie in P_1 und P_2 geteilt wird, dann bilden die Kanten des Voronoi-Diagramms für P, die sowohl an Regionen für Punkte aus P_1 als auch an Regionen für Punkte aus P_2 angrenzen, einen in vertikaler Richtung monotonen, zusammenhängenden Kantenzug.

Berechne den P_1 und P_2 trennenden Kantenzug K, der Teil von $VD(P)$ ist.

Schneide den rechts von K liegenden Teil von $VD(P_1)$ ab. Schneide den links von K liegenden Teil von $VD(P_2)$ ab. Vereinige $VD(P_1)$ und $VD(P_2)$ und K.

K kann aus den Mittelsenkrechten der Punktpaare auf der konvexen Hülle von P_1 und P_2 zusammengebaut werden.

Berechnung des trennenden Kantenzuges K bei gegebenen Voronoi-Diagrammen $VD(P_1)$ und $VD(P_2)$

Ermittle die oberen Tangentialpunkte $p'_1 \in P_1$ und $p'_2 \in P_2$.
Ermittle die unteren Tangentialpunkte $p_1 \in P_1$ und $p_2 \in P_2$.
Sei m die Mittelsenkrechte der Punkte p'_1 und p'_2.
Wähle $k = (x_k, y_k)$ mit $y_k = max_y$, so daß k auf m liegt.
$K = \emptyset$
while $(p'_1 \neq p_1)$ oder $(p'_2 \neq p_2)$
{ ermittle den Schnittpunkt s_1 von m mit $VR(p'_1)$ unterhalb von k
  ermittle den Schnittpunkt s_2 von m mit $VR(p'_2)$ unterhalb von k
  if (s_1 liegt oberhalb von s_2) oder (s_2 existiert nicht)
    then $i = 1$ else $i = 2$
  füge Geradenstück m von k bis s_i zu K hinzu
  sei $p''_i \in P_i$, dessen Voronoi-Region $VR(p''_i)$ in s_i an $VR(p'_i)$ angrenzt
  $p'_i = p''_i$
}
Füge m von k bis $k' = (x'_k, y'_k)$ mit $y'_k = min_y$ und k' liegt auf M zu K hinzu

Die Berechnung von K kann in Zeit $O(|P_1| + |P_2|)$ erfolgen, wenn die konvexen Hüllen von P_1 und P_2 mitberechnet werden.

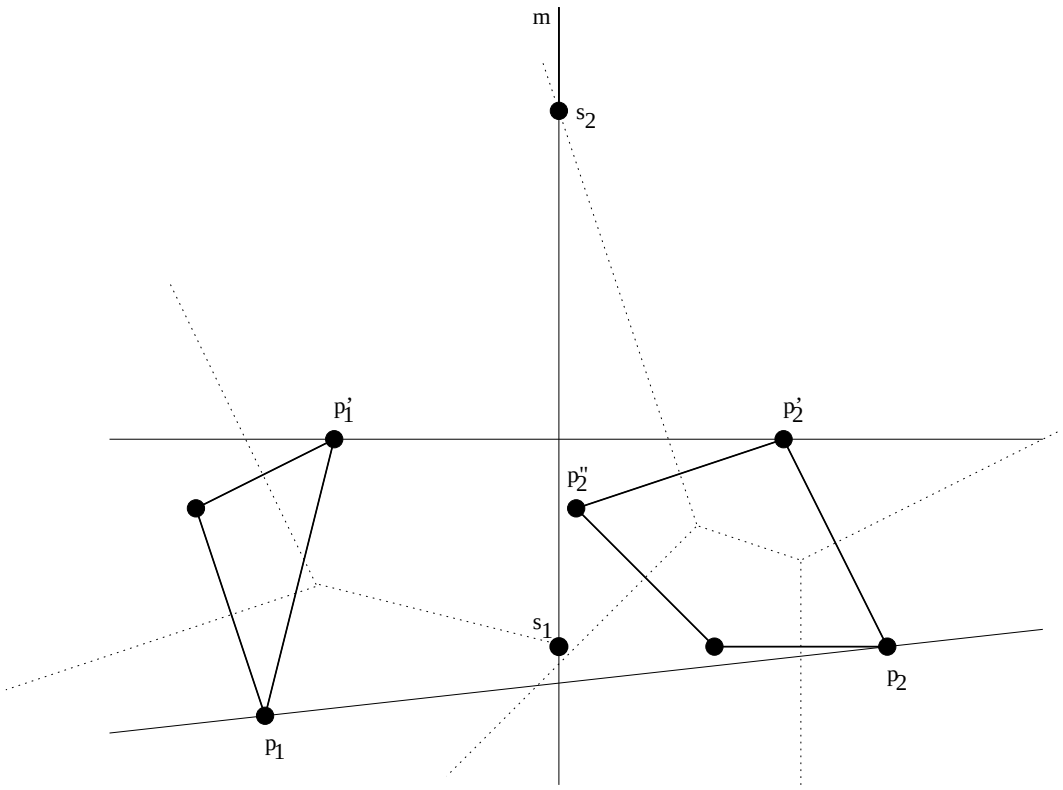
$\Rightarrow$ Das Voronoi-Diagramm kann in Zeit $O(n \cdot log(n))$ berechnet werden.

Lösung von Distanzproblemen:

Dichtestes Punktpaar:

Konstruiere das Voronoi-Diagramm, durchlaufe die doppelt verkettete Kantenliste und ermittle dabei das Minimum der Distanz benachbarter Punkte

$\Rightarrow$ Zeit: $O(n \cdot log(n))$



Alle nächsten Nachbarn:

Konstruiere das Voronoi-Diagramm und durchlaufe die doppelt verkettete Kantenliste, so daß der Reihe nach für jeden Punkt p alle Voronoi-Kanten von $VR(p)$ betrachtet werden.

Dabei wird für jeden Punkt ein nächster Nachbar unter allen Punkten mit benachbarter Voronoi-Region bestimmt.

$\Rightarrow$ Zeit: $O(n \cdot log(n))$

Minimaler spannender Baum:

Konstruiere das Voronoi-Diagramm und die Delaunay-Triangulierung.

$\Rightarrow$ Zeit: $O(n \cdot log(n))$

Vorverarbeitung (Preprocessing) für „Suche nächsten Nachbarn für q“

- Konstruiere das Voronoi-Diagramm.
- Trianguliere die beschränkten Voronoi-Regionen.
- Umgebe die entstehende Triangulation der beschränkten Voronoi-Regionen mit einem Dreieck.

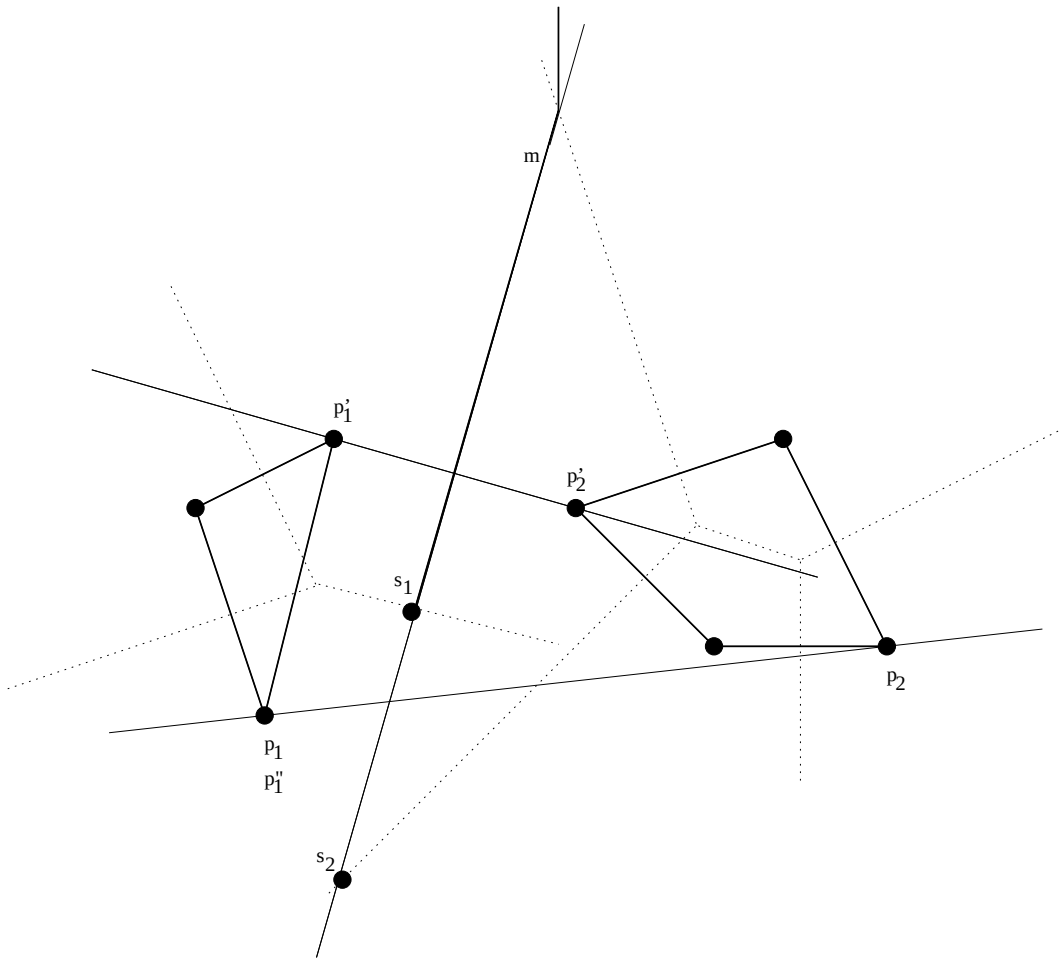


Abbildung 10.16: Berechnung des trennenden Kantenzuges

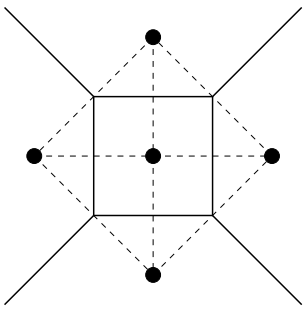


Abbildung 10.17: Berechne einen minimalen spannenden Baum für die Delaunay-Triangulierung $DT(P)$.

Die Differenzregion wird ebenfalls trianguliert.

Die Anzahl der Dreiecke ist proportional zur Anzahl der Voronoi-Kanten, also linear in der Anzahl der Punkte in P.

In Zeit $O(n \cdot log(n))$ konstruierbar mit Scan-Line-Verfahren.

Suche das Dreieck D, das p enthält.

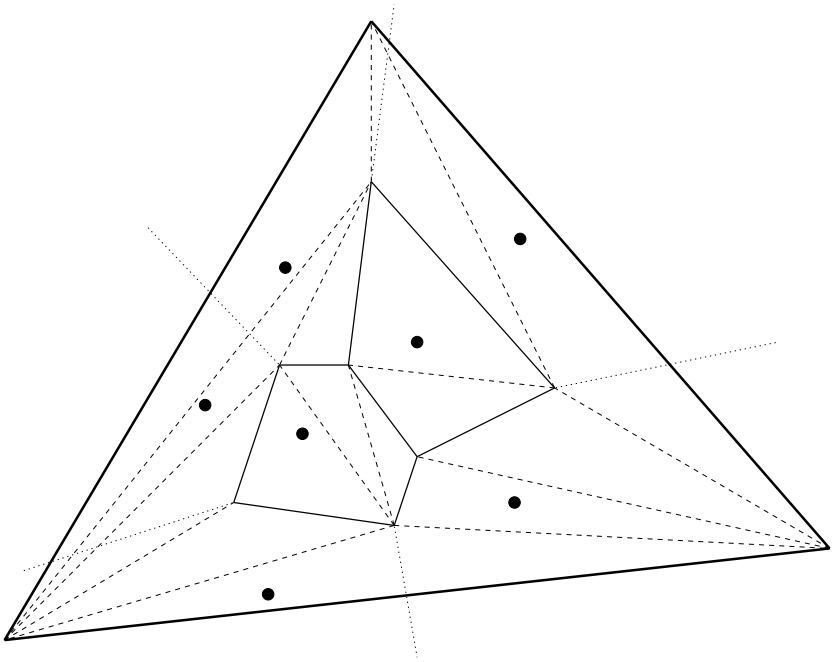


Abbildung 10.18: Preprocessing für „Suche nächsten Nachbarn“

Ist D ein Teil einer beschränkten Voronoi-Region $VR(q)$, dann ist q der nächste Nachbar von p.

Liegt p außerhalb des umschließenden Dreiecks oder in der Differenzregion, dann wird in Zeit $O(n \cdot log(n))$ eine binäre Suche auf den unbeschränkten Voronoi-Regionen durchgeführt.

Gesucht wird das Dreieck D, das p enthält:

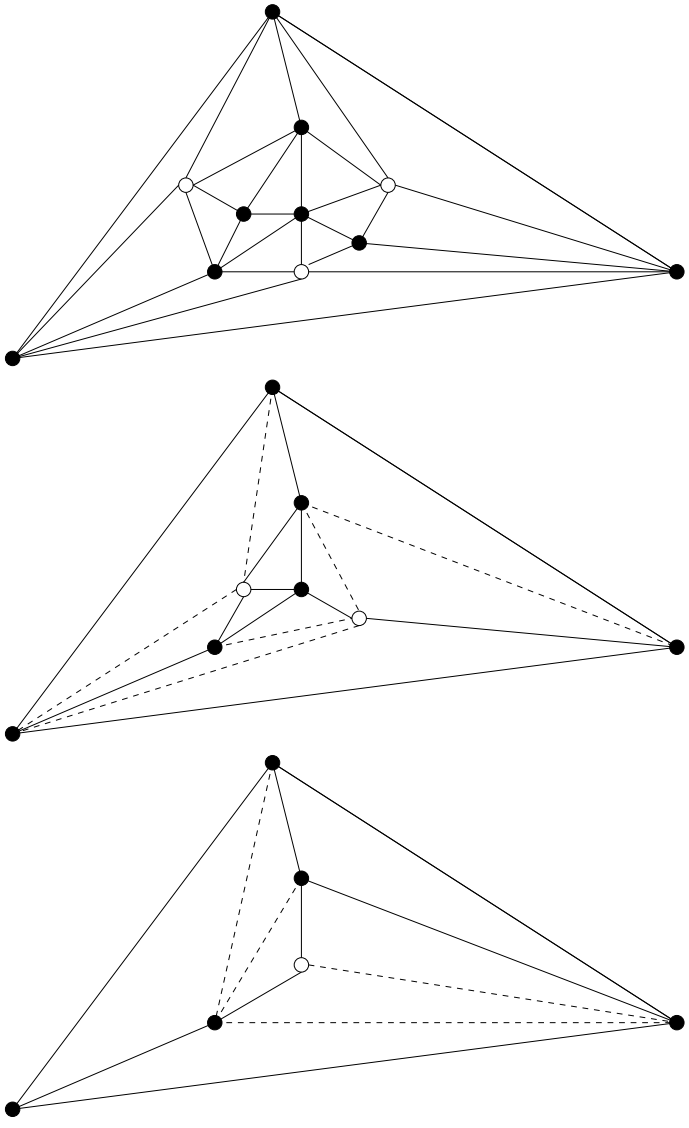
Vergröbere die Triangulation wie folgt:

- Suche unabhängige Punktmengen, die nicht auf dem Rand der Triangulation liegen und nicht miteinander verbunden sind.
- Entferne die Punkte und trianguliere die entstehenden Polygone.
- Wiederhole die Vergröberung bis das umschließende Dreieck übrigbleibt.

Beginne die Suche mit der gröbsten Triangulation.

Führe die Suche bei Erfolg mit der entsprechenden feineren Triangulation weiter, bis D gefunden ist.

Die Laufzeit ist abhängig von der Anzahl der Vergröberungen (möglichst $O(log(n))$ viele) und der Anzahl der Dreiecke zur Triangulation der Polygone



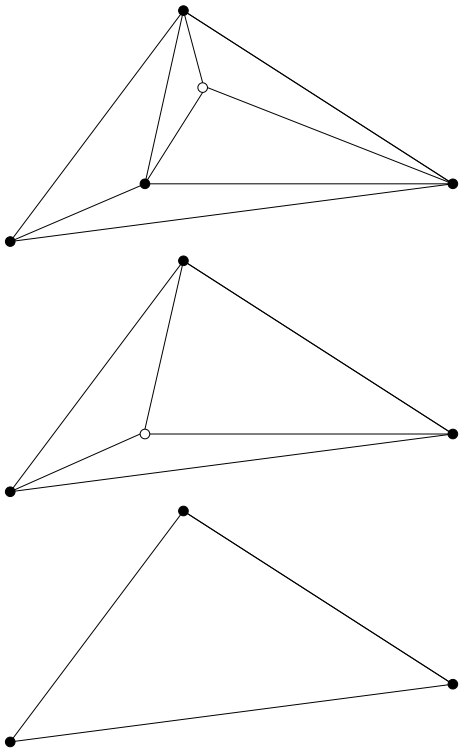


Abbildung 10.19: Triangulierungsvergröberung für „Suche nächsten Nachbarn“

(möglichst $O(1)$ viele). Beide Werte sind abhängig von der Wahl der unabhängigen Knoten.

Idee von Kirkpatrick:

Entferne eine unabhängige Menge von Knoten, die jeweils einen Grad kleiner als 12 haben. In einer Triangulation mit n Punkten gibt es höchstens $3 \cdot n - 6$ Kanten

$\Rightarrow$ Die Summe der Knotengrade ist $< 6 \cdot n$.

$\Rightarrow$ Mindestens die Hälfte der Knoten haben einen Grad < 12.

Die Anzahl der zu entfernenden Punkte ist $v \geq \lfloor \frac{1}{12} \cdot (\frac{n}{2} - 3) \rfloor \Rightarrow \frac{v}{n} \geq \frac{1}{24} - \frac{1}{4 \cdot n}$

Für genügend großes n (z.B. $n \geq 12$ und $\frac{v}{n} \geq \frac{1}{48}$) wird immer ein fester (wenn auch kleiner) Bruchteil aller Punkte in einem Vergröberungsschritt entfernt. Folglich ist die Anzahl der Vergröberungen in $O(log(n))$.

$\Rightarrow$ Die „Nächster Nachbar“-Anfragen können in Zeit $O(log(n))$ mit einer Vorverarbeitungszeit von $O(n \cdot log(n))$ gelöst werden.

11 Planare Graphen

11.1 Definitionen und Einführung

Definition: Ein ungerichteter Graph $G = (V, E)$ mit $E \subseteq \big\{\{u, v\}\big|u \neq v,\ u, v \in V\big\}$ ist planar, wenn er so in der Ebene gezeichnet werden kann, daß jeder Knoten u ein Punkt $p(u)$ ist und jede Kante $\{u, v\}$ eine Linie ist, die die beiden Punkte $p(u)$ und $p(v)$ verbindet, so daß sich keine zwei Linien kreuzen.

Die Punkte der Knoten und die Linien der Kanten bilden eine planare Einbettung von G.

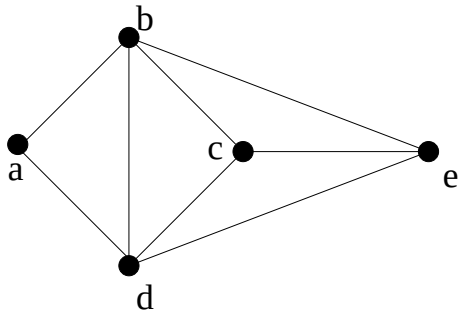
Ein Face ist eine maximale zusammenhängende Fläche der planaren Einbettung, die nicht durch eine Kantenlinie durchkreuzt wird. Genau ein Face besitzt eine unendlich große Fläche, nämlich das äußere Face. Die Kontur eines Faces ist ein Kreis in G, den wir Fenster nennen.

Bemerkung: Ein Fenster muss nicht umbedingt eine einfacher Kreis sein.

Beispiel: Sei $G = (V, E)$ ein Graph mit

$V = \{a, b, c, d, e\}$ und $E = \big\{\{a, b\}, \{b, d\}, \{b, c\}, \{b, e\}, \{c, e\}, \{c, d\}, \{d, e\}, \{a, d\}\big\}$

Eine planare Einbettung für G sieht zum Beispiel so aus:



Der vollständige Graph mit 5 Knoten (der K_5, links in Abbildung 11.20) und der vollständige bipartite Graph mit zweimal 3 Knoten (der $K_{3,3}$, rechts in Abbildung 11.20) sind nicht planar.

Definition: Ein Graph G ist einer Erweiterung bzw. Unterteilung von $H = (V_H, E_H)$, wenn G aus H durch Einfügen von Knoten vom Grad 2 auf den Kanten entstehen kann.

Definition: Zwei Graphen G und H sind homöomorph, wenn sie beide eine Erweiterung eines Graphen J sind.

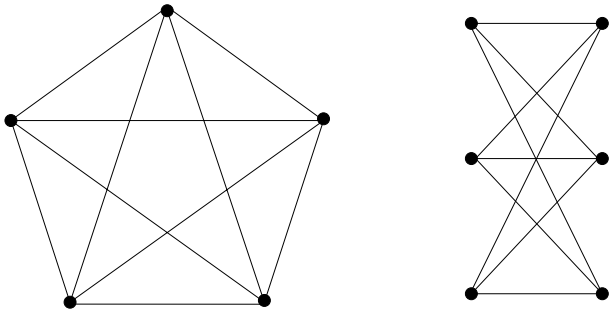


Abbildung 11.20: Zwei nicht planare Graphen

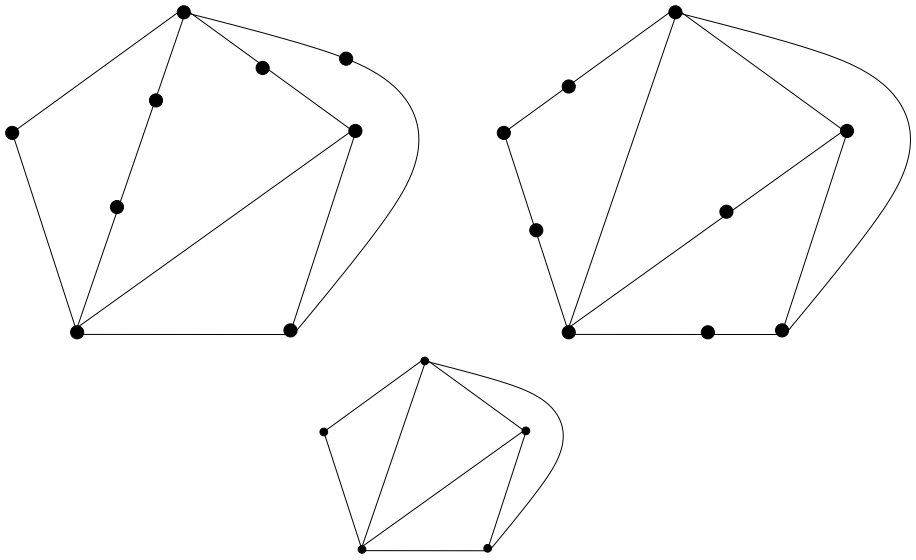


Abbildung 11.21: Zwei homöomorphe Graphen (groß) und der Graph, aus dem beide enstanden sind (klein)

Satz: [Kuratowski's Theorem] Ein Graph G ist genau dann nicht planar, wenn G einen Teilgraphen G' hat, der eine Erweiterung des K_5 oder $K_{3,3}$ ist.

Definition: Seien R_1 und R_2 zwei planare Einbettungen eines Graphen G. Wir sagen, R_1 und R_2 sind äquivalent, wenn jedes Fenster in R_1 ein Fenster in R_2 ist und umgekehrt. (Die Orientierung der Fenster in R_2 darf umgekehrt zur Orientierung der Fenster in R_1 sein.)

Definition: Sei $G = (V, E)$ ein Graph und $C = (V_C, E_C)$ ein einfacher Kreis in G mit Knotenmenge V_C und Kantenmenge E_C.

1. Ein Graph $B = \big(\{u, v\}, \{\{u, v\}\}\big)$ mit $u, v \in V_C$ und $\{u, v\} \in E \setminus E_C$ heißt Brücke für den Kreis C in G.

2. Sei (V', E') eine Zusammenhangskomponente in dem durch $V \setminus V_C$ induzierten Teilgraphen $G|_{V \setminus V_C}$ von G. Dann ist der Graph

$$B = (V_B, E_B)$$

mit Knotenmenge

$$V_B = V' \cup \{u \in V_C \mid \{u, v\} \in E, v \in V'\}$$

und der Kantenmenge

$$E_B = E' \cup \{\{u, v\} \in E \mid u \in V', v \in V_B\}$$

ebenfalls eine Brücke für C in G, falls er mindestens zwei Knoten aus C enthält.

Satz: Sei G ein Graph. Wenn es eine planare Einbettung R für G gibt, sodass ein Fenster in R mehr als eine Brücke hat, dann kann G durch Herausnahme von zwei Knoten separiert werden.

Beweis: Sei F ein Fenster mit zwei Brücken B_1 und B_2. Seien V_1 und V_2 die Mengen der Knoten aus dem Fenster F, die auch zur Brücke B_1 bzw. Brücke B_2 gehören.

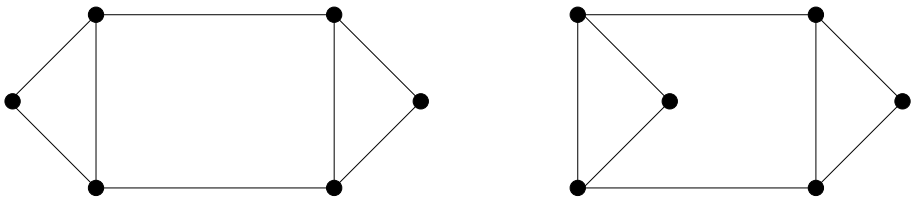


Abbildung 11.22: Zwei nicht äquivalente Einbettungen für einen Graphen

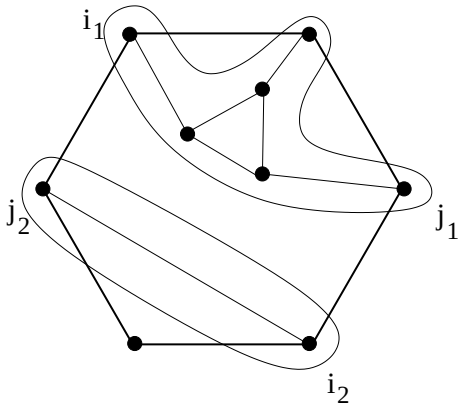


Abbildung 11.23: Für das äußere Fenster gibt es zwei Brücken in G

Starte bei einem beliebigen Knoten im Kreis F und durchlaufe den Kreis in einer vorgegebenen Richtung bis jeder Knoten mehrfach besucht wurde.

Da G planar und F in R_1 ein Fenster ist, können die beiden Brücken nicht ineinander verschränkt sein. Es gibt also keine 4 Knoten $u_1, u_3 \in V_1$, $u_2, u_4 \in V_2$, die in der Reihenfolge u_1, u_2, u_3, u_4 oder umgekehrt besucht werden. Sei u_{a_1}, u_{a_2} der erste Knoten aus V_1 bzw. V_2, der nach einem Knoten aus V_2 bzw. V_1 besucht wird, und sei u_{e_1}, u_{e_2} der letzte Knoten aus V_1 bzw. V_2, der vor einem Knoten aus V_2 bzw. V_2 besucht wird. Dann separieren die beiden Knoten u_{a_1}, u_{e_1} und die beiden Knoten u_{a_2}, u_{e_2} den Graphen.

Satz: Sei G ein dreifach zusammenhängender Graph. Dann sind alle planaren Einbettungen von G äquivalent.

Beweis: Angenommen, F ist ein Fenster in einer planaren Einbettung R_1 für G, sodass eine zweite planare Einbettung R_2 existiert, in der der Kreis F kein Fenster ist.
Da F in R_2 kein Fenster ist, hat F mindestens zwei Brücken (eine innere und eine äußere). Da F in R_1 ein Fenster mit ebenfalls zwei Brücken ist, folgt aus obigem Satz, dass G durch Herausnahme von zwei Knoten separiert werden kann. Dies widerspricht der Voraussetzung, dass G dreifach zusammenhängend ist.

Satz: Sei $G = (V, E)$ ein zusammenhängender planarer Graph. Für jede planare Einbettung R für G gilt:

$$|V| + f - |E| = 2,$$

wobei f die Anzahl der Fenster in R ist.

Beweis: Betrachte eine planare Einbettung R für G und einen Spannbaum $S = (V', E')$ für G. Dann gilt für den Spannbaum $f' = 1$, $|V'| = |E'| + 1$ und somit $|V'| + f' - |E'| = 2$. Mit jeder Kante, die man hinzufügt, erhöht sich die Anzahl der Fenster ebenfalls um 1.

Bemerkung: Somit haben alle zusammenhängenden Graphen mit n Knoten und m Kanten in ihren planaren Einbettungen die gleiche Anzahl von Fenster.

Satz: Sei $G = (V, E)$ ein zusammenhängender planarer Graph mit $|V| \geq 3$. Dann gilt

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

Beweis: Sei R eine planare Einbettung für G. Jedes Fenster hat mindestens 3 Kanten, und jede Kante ist an höchstens 2 Fenstern beteiligt. Also ist $3f \leq 2|E|$ und wegen $|E| = |V| + f - 2$ ist $|E| \leq |V| + \frac{2}{3}|E| - 2$ und somit

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

Bemerkung: Jeder planare Graph mit mindestens 4 Knoten hat mindestens 4 Knoten vom Grad ≤ 5.

Bemerkung: Ein Graph $G = (V, E)$ ist k-färbbar, wenn es eine Abbildung $f : V \to \{1, \ldots, k\}$ gibt, so daß für jede Kante $\{u, v\}$ gilt:

$$f(u) \neq f(v)$$

Satz: Jeder planare Graph $G = (V, E)$ ist 6-färbbar.

Beweis: Induktion über die Anzahl der Knoten der Graphen. Ein planarer Graph mit ≤ 6 Knoten ist immer 6-färbbar.

Sei G ein planarer Graph mit mehr als 6 Knoten und u ein Knoten vom Grad ≤ 5. Laut Induktionsvoraussetzung ist

$$G' = G \setminus \{u\} = \big(V \setminus \{u\}, E \setminus \big\{\{u, v\} \big| v \in V\big\}\big)$$

6-färbbar. Färbe u nach der Färbung von G' mit einer Farbe, die nicht von seinen Nachbarn benutzt wird.

$\Rightarrow G$ ist 6-färbbar.

Satz: Jeder planare Graph $G = (V, E)$ ist 5-färbbar.

Beweis: Induktion über die Anzahl der Knoten des Graphen. Ein planarer Graph mit ≤ 5 Knoten ist immer 5-färbbar.

Sei G ein planarer Graph mit mehr als 5 Knoten und u ein Knoten mit Grad ≤ 5. Hat u einen Grad ≤ 4, dann färbe $G \setminus \{u\} = (V', E')$ mit 5 Farben und gebe u eine von seinen Nachbarn nicht benutzte Farbe.

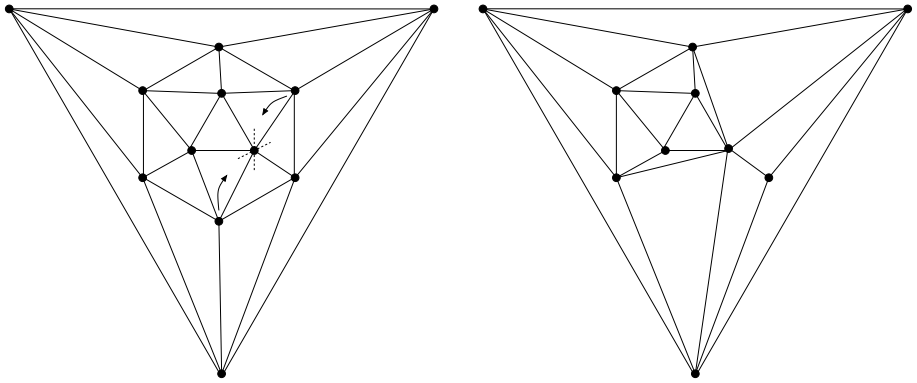


Abbildung 11.24: Zum Beweis der Fünffärbbarkeit

Hat u den Grad 5, dann gibt es zwei Nachbarn v, w von u, die nicht adjazent sind, andernfalls wäre der K_5 Teilgraph von G. Sei

$$G_{v \to w} := \Big(V' \setminus \{v\}, E' \setminus \big\{ \{v, v'\} \big| v' \in V' \big\} \cup \big\{ \{w, v'\} \big| \{v, v'\} \in E' \big\} \Big)$$

der Graph nach Löschen von u und der Kontraktion von v mit w.

Der Graph $G_{v \to w}$ ist planar, da G eine planare Einbettung besitzt, in der v und w auf einem Fenster liegen.

Färbe den planaren Graphen $G_{v \to w}$ mit 5 Farben. Dies ist laut Induktionsvoraussetzung möglich. Mache anschließend die Kontraktion von v und w rückgängig. Die beiden Knoten erhalten die gleiche Farbe. Füge u wieder ein und gebe u eine von seinen Nachbarn nicht benutzte Farbe. Da die Nachbarn von u höchstens 4 verschiedene Farben haben, ist der Graph G 5-färbbar.

Bemerkung: Es gibt planare Graphen, in denen jeder Knoten den Grad ≥ 5 hat (vgl. Abbildung 11.24).

Satz: Jeder planare Graph $G = (V, E)$ ist 4-färbbar.

Satz: Nicht jeder planare Graph ist 3-färbbar. Das Entscheidungsproblem, ob ein planarer Graph 3-färbbar ist, ist NP-vollständig.

Beweis: Der K_4 ist planar und offensichtlich nicht 3-färbbar.
Reduktion von 3-Färbbarkeit allgemeiner Graphen. Sei $H = (V_H, E_H)$ der Graph in Abbildung 11.25. Der Graph H hat die folgenden Eigenschaften:

1. Für jede 3-Färbung $f : V_H \to \{1, 2, 3\}$ für H gilt:

$$f(x) = f(x') \text{ und } f(y) = f(y')$$

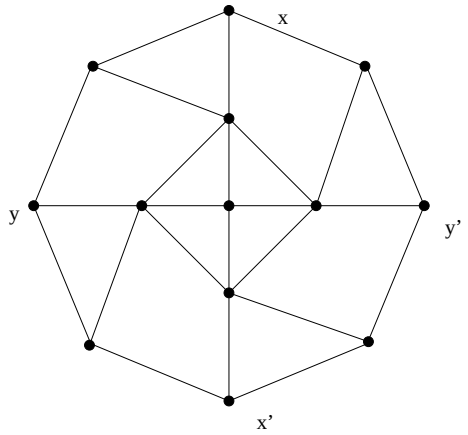


Abbildung 11.25: Graph H im Beweis von Satz **??**

2. Es gibt eine 3-Färbung $f : V_H \to \{1, 2, 3\}$ für H mit

$$f(x) = f(x') = f(y) = f(y')$$

3. Es gibt eine 3-Färbung $f : V_H \to \{1, 2, 3\}$ für H mit

$$f(x) = f(x') \neq f(y) = f(y')$$

Betrachte eine nicht kreuzungsfreie Einbettung für einen beliebigen Graphen G, so daß sich in einem Punkt höchstens zwei Kantenlinien kreuzen. Betrachte zwei Kanten $\{u, u'\}$ und $\{v, v'\}$ und den Kreuzungspunkt p, so daß es zwischen $p(u)$ und p sowie zwischen $p(v)$ und p keine weiteren Kreuzungen gibt. Füge nun eine Kopie von H hinzu, identifiziere Knoten x mit u und y mit v, streiche die Kanten $\{u, u'\}$ und $\{v, v'\}$ und füge die Kanten $\{x', u'\}$ und $\{y', v'\}$ hinzu. Benutze die planare Einbettung von H, um für den neuen Graphen eine Einbettung zu konstruieren, die eine Kreuzung weniger enthält.

1. Die Transformation von G in einen äquivalenten Graphen G' benötigt polynomielle Zeit.

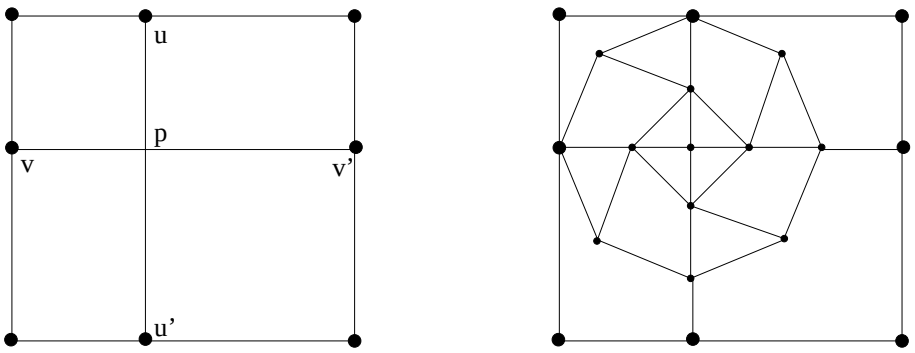


Abbildung 11.26: Ersetzung eines Kreuzungspunktes durch H

2. Der neue Graph G' ist genau dann 3-färbbar, wenn der initiale Graph G 3-färbbar ist.

Satz: Planarität kann in linearer Zeit getestet werden.

Beweise:

1967: International Symposium on Theory of Graphs, Even, Cederbaum, Lempel, Algorithms for Planarity Testing of Graphs

1974: Journal of the ACM (JACM), Hopcroft, tarjan, Efficient Planarity Testing

Definition: Ein planarer Graph ist maximal planar, wenn keine Kante hinzugefügt werden kann, ohne seine planare Eigenschaft zu zerstören.

Für die folgende Definition betrachten wir ausnahmsweise Graphen mit Schleifen $\{u, u\}$, $u \in V$, und Mehrfachkanten $\big\{\{u, v\}, \{u, v\}\big\}$. Dies ist möglich durch die Einführung von Multimengen.

Definition: Sei G ein ungerichteter zusammenhängender Graph und $E' \subseteq E$ eine Kantenmenge, so daß $G' = (V, E \setminus E')$ nicht zusammenhängend ist, aber für jede echte Teilmenge $E'' \subset E'$ der Graph $G'' = (V, E \setminus E'')$ zusammenhängend ist. Eine solche Kantenmenge nennen wir minimale Kantenseparationsmenge.

Ein Graph $G_2 = (V_2, E_2)$ ist ein dualer Graph eines zusammenhängenden Graphen $G_1 = (V_1, E_1)$, wenn es eine Bijektion $f : E_1 \to E_2$ gibt, so daß für jede minimale Kantenseparationsmenge $E' \subseteq E_1$ für G_1 die Menge $\{f(e) | e \in E'\}$ in G_2 die Kantenmenge eines einfachen Kreises ist.

Bemerkung: Zusammenhängende planare Graphen haben immer einen dualen Graphen.

Konstruktionsschema für den geometrischen dualen Graphen: Sei G ein planarer Graph. Betrachte eine planare Einbettung für G. Erzeuge einen Knoten für jedes Face. Betrachte für jede Kante in G die beiden Faces (die beiden Faces können auch gleich sein) und verbinde die entsprechenden Knoten.

Bemerkung: Jeder duale Graph eines zusammenhängenden planaren Graphen ist planar.

Satz: Ein zusammenhängender Graph hat genau dann einen dualen Graphen, wenn er planar ist.

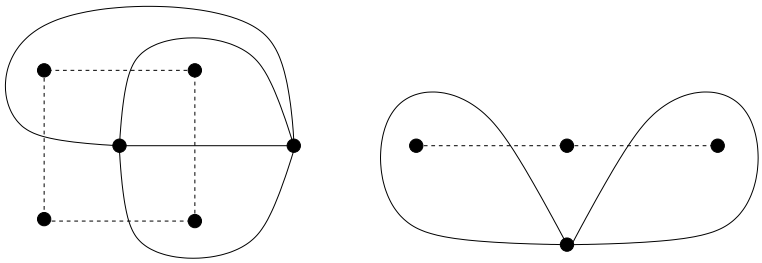


Abbildung 11.27: Beispiele für duale Graphen

Definition: Die Kreuzungszahl $v(G)$ eines Graphen G ist die kleinste Anzahl paarweiser Überschneidungen von Linien, die beim Zeichnen von G in der Ebene auftreten kann.

Bemerkung: $v(G) = 0 \Leftrightarrow G$ ist planar
Die Berechnung der Kreuzungzahl ist ein NP-vollständiges Problem.

Problem: CROSSING NUMBER
Gegeben: Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$
Frage: Ist $v(G) \leq k$?

Die Kreuzungszahl hängt nicht von der Zahl der Kanten ab, die entfernt werden müssen, damit der Graph planar wird. Der Graph in Abbildung 11.28 ist ohne die fett gezeichnete Kante planar. Seine Kreuzungszahl ist aber wesentlich größer als 1, da der Graph 3-fach zusammenhängend ist und somit alle planaren Einbettungen äquivalent sind.

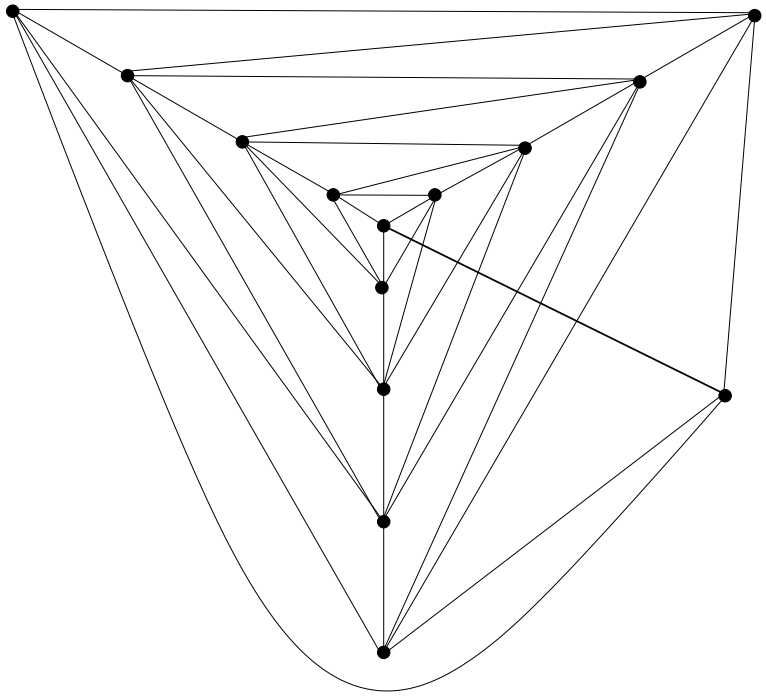


Abbildung 11.28: Die Kreuzungszahl kann beliebig groß werden

11.2 Außenplanare Graphen

Definition: Ein planarer Graph ist außenplanar, wenn es eine planare Einbettung gibt, so daß alle Knoten auf einem Fenster liegen, z.B. dem äußeren Fenster.

Bemerkung: Außenplanarität kann in linearer Zeit getestet werden. Sei $G = (V, E)$ ein Graph und $G' = \Big(V \cup \{u\}, E \cup \big\{\{u, v\} \big| v \in V\big\}\Big)$ für einen neuen Knoten $u \notin V$. Der Graph G ist genau dann außenplanar, wenn G' planar ist (vgl. Abbildung 11.29).

Definition: Ein außenplanarer Graph ist *maximal außenplanar*, wenn keine Kante hinzugefügt werden kann, ohne seine Außenplanarität zu zerstören.

Bemerkung: Die planare Einbettung eines maximal planaren Graphen ist eine Triangulation der Sphäre.
Die planare Einbettung eines maximal außenplanaren Graphen ist eine Triangulation eines Polygons.

Satz: Sei $G = (V, E)$ ein maximal außenplanarer Graph mit mindestens 3 Knoten und R eine planare Einbettung für G, so daß alle Knoten auf dem äußeren Face liegen. Dann gibt es in R genau $|V| - 2$ innere Faces.

Beweis: Induktion über die Knotenzahl.

- $|V| = 3$ Die Aussage gilt offensichtlich.
- $|V| > 3$ Sei $u_1, \ldots, u_n$ die Reihenfolge der Knoten auf dem äußere Fenster in der Einbettung von G. Da G mehr als 3 Knoten hat und maximal außenplanar ist, gibt es mindestens eine Kante $\{u_i, u_j\}$, die nicht zum äußeren Fenster gehört. Sei $i < j$, $U = \{u_1, \ldots, u_i, u_j, \ldots, u_n\}$, $U' = \{u_i, \ldots, u_j\}$, $|U| = m$ und $|U'| = m'$. Dann sind die durch U und U' induzierten Graphen $G|_U$

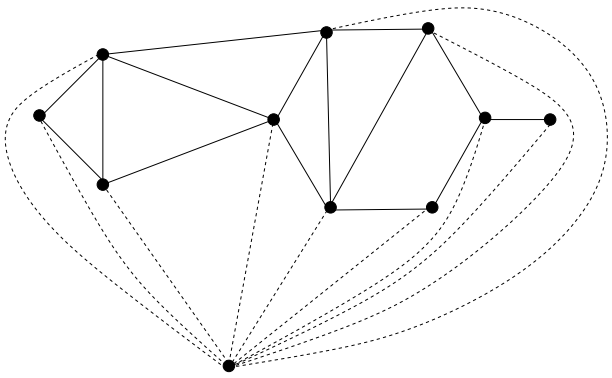


Abbildung 11.29: Ein außenplanarer Graph mit einem zusätzlichen Knoten wie in der Bemerkung

und $G|_{U'}$ ebenfalls maximal außenplanarer. Da $m \geq 3$ und $m' \geq 3$, gilt nach Induktionsvoraussetzung, dass $G|_U$ genau $m-2$ und $G|_{U'}$ genau $m'-2$ innere Faces hat. Da $n = m' + m'' - 2$, hat also die Einbettung von G genau $m - 2 + m' - 2 = n - 2$ Faces.

Folgerung: Sei $G = (V, E)$ ein außenplanarer Graph, dann gilt:

1. $|E| \leq 2 \cdot |V| - 3$
2. Mindestens zwei Knoten haben einen Grad ≤ 2.
3. Mindestens drei Knoten haben einen Grad ≤ 3.

Bemerkung: Der K_4 und der $K_{2,3}$ ist nicht außenplanar.

Satz: [Chartrand, Harrary] Ein Graph G ist genau dann außenplanar, wenn G keinen Teilgraphen enthält, der eine Erweiterung des K_4 oder $K_{2,3}$ ist.

Bemerkung: Ein Graph $G = (V, E)$ ist 2-*reduzierbar*, wenn

1. $E = \emptyset$ oder
2. G hat einen Knoten u vom Grad 1 und $G \setminus \{u\}$ ist 2-reduzierbar oder
3. G hat einen Knoten u vom Grad 2 und $G \setminus \{u\}$ mit der zusätzlichen Kante $\{v, w\}$ (falls sie noch nicht existiert) ist 2-reduzierbar, wobei v, w die beiden Nachbarn von u sind.

Satz: Jeder außenplanare Graph ist 2-reduzierbar.

Beweis: Ein außenplanarer Graph hat mindestens zwei Knoten vom Grad ≤ 2. Da der Graph nach einem Reduktionsschritt außenplanar bleibt, folgt die Behauptung.

Lemma: Sei $G = (V, E)$ ein 2-reduzierbarer Graph. Dann ist $|E| \leq 2 \cdot |V| - 3$.

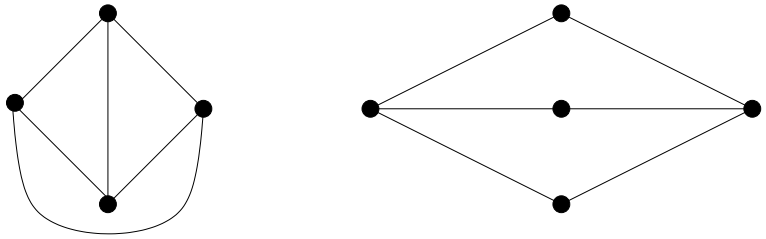


Abbildung 11.30: Der K_4 und der $K_{2,3}$ sind nicht außenplanar.

Beweis: Reduziere den Graphen G, bis zwei Knoten und maximal eine Kante übrig bleibt. Für das Ergebnis gilt $|E| \leq 2 \cdot |V| - 3$. In jedem Schritt wurden ein Knoten und maximal zwei Kanten entfernt.

Satz: [Dirac, 1953] Jeder zweifach zusammenhängende Graph, in dem jeder Knoten einen Grad ≥ 3 hat, hat einen Teilgraphen, der eine Erweiterung des K_4 ist.

Satz: Ein Graph $G = (V, E)$ ist genau dann 2-reduzierbar, wenn er keinen Teilgraphen enthält, der eine Erweiterung des K_4 ist.

Beweis:

- $\Leftarrow$: Wenn G einen Teilgraphen enthält, der eine Erweiterung des K_4 ist, dann ist G offensichtlich nicht 2-reduzierbar.
- $\Rightarrow$ Durch Widerspruch. Sei G ein nicht 2-reduzierbarer Graph mit einer minimalen Anzahl von Knoten, der keinen Teilgraphen enthält, der eine Erweiterung des K_4 ist. Alle Knoten haben einen Grad ≥ 3. Wenn G zweifach zusammenhängend ist, dann widerspricht unsere Annahme Diracs Theorem. Wenn G nicht zweifach zusammenhängend ist, dann gibt es einen kleineren Graphen, der nicht 2-reduzierbar ist und einen Teilgraphen enthält, der keine Erweiterung des K_4 ist. Widerspruch!

Definition: Für einen Graphen $G = (V, E)$ und einen Knoten $u \in V$ vom Grad ≤ 2 sei G_u der Graph ohne u, in dem die beiden Nachbarn von u mit einer Kante verbunden wurden, falls u zwei Nachbarn hatte und diese Kante noch nicht existierte.

Lemma: Sei G ein Graph und u ein Knoten vom Grad ≤ 2. Es gilt: G_u ist genau dann 2-reduzierbar, wenn G 2-reduzierbar ist.

11.3 Test auf 2-Reduzierbarkeit

Bemerkung. Bei einem 2-Reduktionsschritt muß getestet werden, ob die beiden Nachbarknoten von u bereits durch ein Kante verbunden sind oder nicht. Daher kommt man nicht ohne weiteres auf einen Algorithmus mit linearer Laufzeit.

Idee: Füge beim Entfernen eines Knotens u die Kante zwischen seinen beiden Nachbarn v, w erst dann ein, wenn einer der beiden Knoten v, w den Grad 2 bekommt.

Sei $G = (V, E)$ ein Graph mit $|V| \geq 4$.
Sei $N(u)$ die Liste der Nachbarn von u.
Sei $N'(u)$ die Liste der noch nicht berücksichtigten Nachbarn von u.
Die Funktion INSERT(u, v) fügt Knoten u in $N(v)$ und Knoten v in $N(u)$ ein.
Die Funktion INSERT'(u, v) fügt Knoten u in $N'(v)$ und Knoten v in $N'(u)$ ein.
Die Funktion DELETE(u, v) entfernt u aus $N(v)$ und v aus $N(u)$.
Die Funktion DELETE'(u, v) entfernt u aus $N'(v)$ und v aus $N'(u)$.
Die Funktionen DELETE(u, v) und DELETE'(u, v) verändern dabei auch die Kantenmenge E.

Test auf 2-Reduzierbarkeit
Eingabe: Graph $G = (V, E)$ mit $|V| \geq 4$
Ausgabe: <ja>, falls G 2-reduzierbar ist
<Nein> sonst

REDUCE(u)
{ case $|N(u)|$ of
  0: ;
  1: sei $u_1 \in N(u)$;
     DELETE(u, u_1);
     if $|N(u_1)| == 2$ then $M = M \cup \{u_1\}$;
  2: seien $u_1, u_2 \in N(u)$;
     DELETE(u, u_1);
     DELETE(u, u_2);
     INSERT'(u_1, u_2);
     if $|N(u_1)| == 2$ then $M = M \cup \{u_1\}$;
     if $|N(u_2)| == 2$ then $M = M \cup \{u_2\}$;
}

MOVE_EDGE(u)
{ Sei $(u_1 \in N'(u))$;
  DELETE'(u, u_1);
  if $u_1 \notin N(u)$ then INSERT(u, u_1);
  if $|N(u)| \leq 2$ then $M = M \cup \{u\}$;
  if $|N(u_1)| > 2$ und $u_1 \in M$ then $M = M \setminus \{u_1\}$;
}

TEST_OF_2-REDUCIBILITY
{ if $|E| > 2 \cdot |V| - 3$ then return(<Nein>);
  $M = \emptyset$; (M ist die Menge aller Knoten vom Grad ≤ 2)
  for all $u \in V$

```
  { if |N(u)| ≤ 2 then M = M ∪ {u};
    N'(u) = ∅;
  }
  while M ≠ ∅
  { sei u ∈ M;
    M = M \ {u};
    if |N'(u)| > 0 then MOVE_EDGE(u);
    else REDUCE(u);
  }
  if |E| == 0 then return(<Ja>);
  else return(<Nein>);
}
```

Bemerkung. Dieser Algorithmus hat eine lineare Laufzeit.

Beispiel. Betrachte Abbildung 11.31. Die mit $*$ markierten Knoten sind in der Menge M, die mit $\rightarrow$ markierten Knoten werden als nächstes betrachtet. Die gestrichelten Kanten sind die, die durch $N'()$ definiert werden.

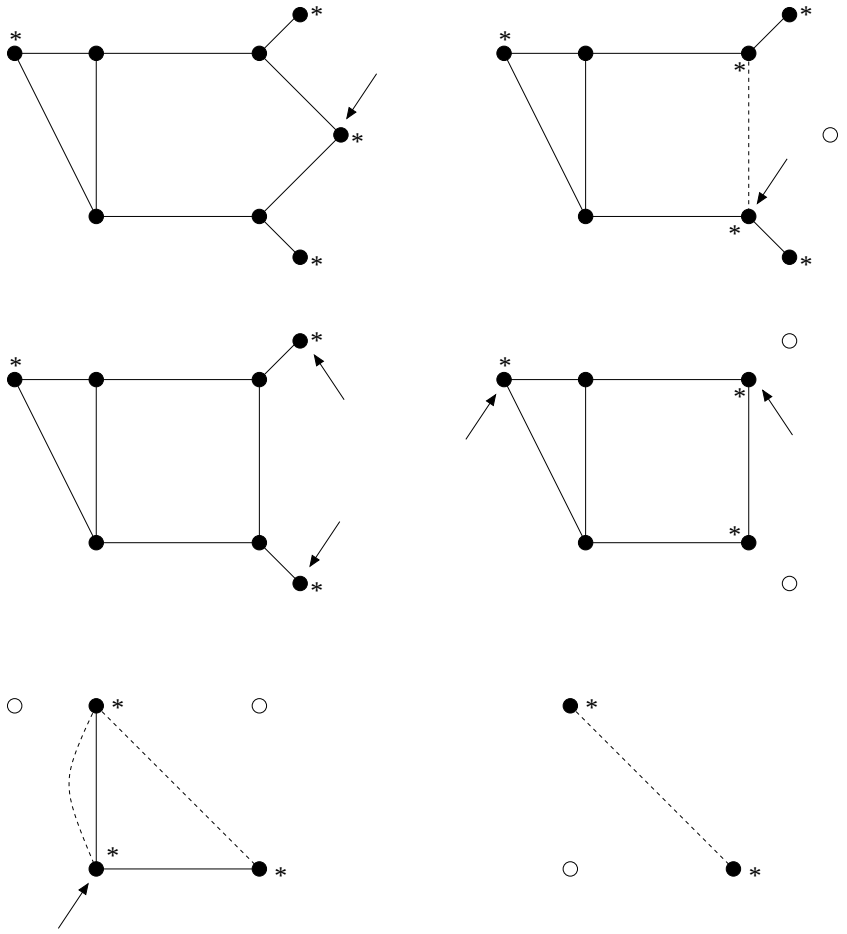


Abbildung 11.31: Beispiel für 2-Reduktion

11.4 Test auf Außenplanarität

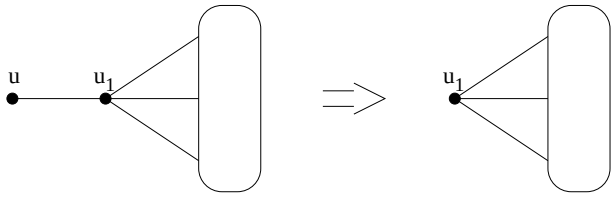
Modifiziere den Test auf 2-Reduzierbarkeit. Wir markieren die Kanten wie folgt:

`out:` äußere Kanten
`cross:` innere Kanten
`bridge:` Verbindungskanten

Initial sind alle Kanten mit `out` markiert.
Sei $A = \{$`cross, out, bridge`$\}$ und $B = \{$`cross, out`$\}$ und col(e) die Markierung der Kante e.
Bei dem Algorithmus werden folgende Reduktionsregeln betrachtet:

Fall 1: $|N(u)| = 0$: entferne u

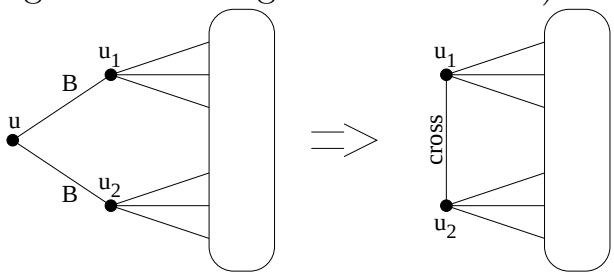
Fall 2: $N(u) = \{u_1\}$ und $\text{col}(\{u, u_1\}) \in A$



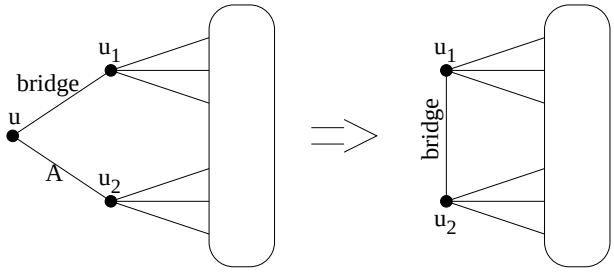
Fall 3: $N(u) = \{u_1, u_2\}$

Fall 3.1: u_1 und u_2 sind nicht adjazent

Fall 3.1.1: Keine der beiden Kanten $\{u, u_1\}$ und $\{u, u_2\}$ ist mit `bridge` markiert (Kantenmarkierung A bzw. B bedeutet, daß diese Kante eine Markierung aus der Menge A bzw. B hat.)

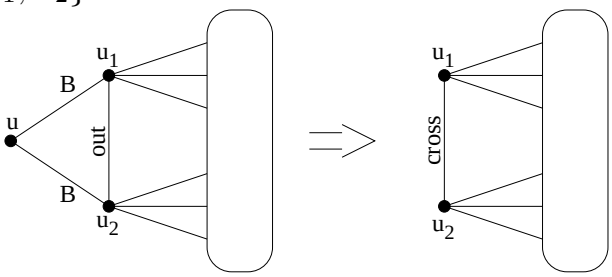


Fall 3.1.2: Mindestens eine der beiden Kanten $\{u, u_1\}$ und $\{u, u_2\}$ ist mit `bridge` markiert

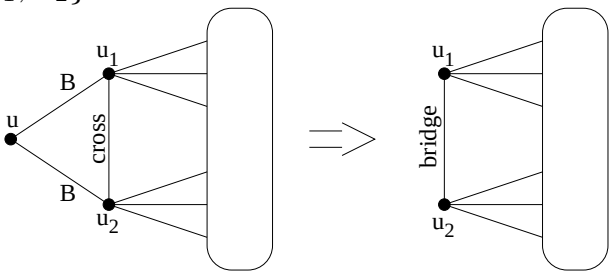


Fall 3.2: u_1 und u_2 sind adjazent

Fall 3.2.1: Kante $\{u_1, u_2\}$ ist mit `out` markiert



Fall 3.2.2: Kante $\{u_1, u_2\}$ ist mit `cross` markiert



Satz 11.1. *Ein Graph G ist genau dann außenplanar, wenn er sich mit diesen 2-Reduktionsregeln vollständig abbauen läßt.*

Bemerkung. Mit diesen Markierungen kann Außenplanarität wie die 2-Reduzierbarkeit in linearer Zeit $O(|V|)$ getestet werden.

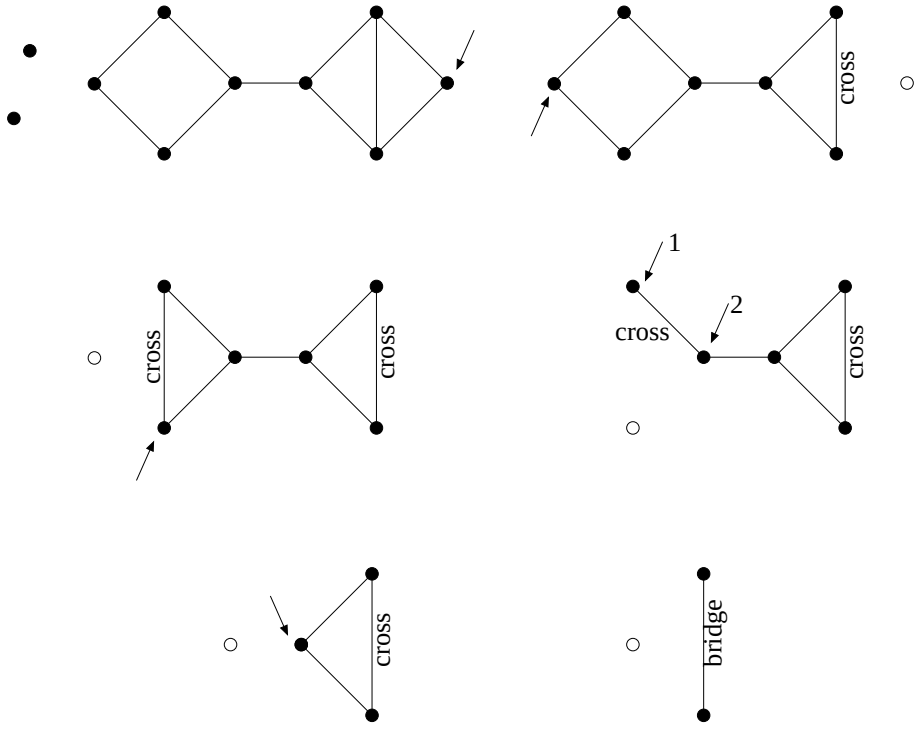


Abbildung 11.32: Beispiel für einen erfolgreichen Außenplanaritätstest, da der Graph komplett abgebaut werden kann

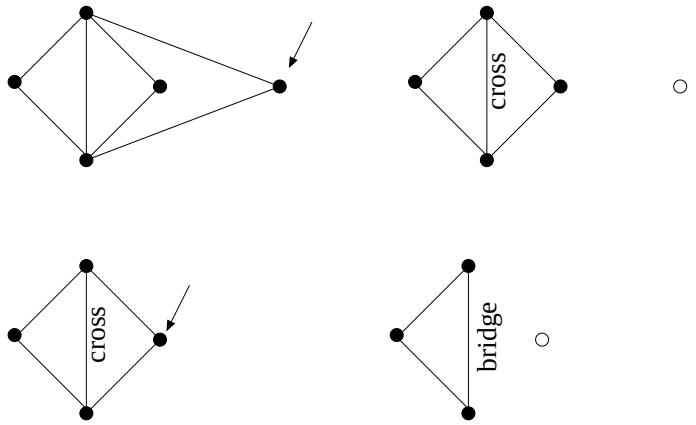


Abbildung 11.33: Beispiel für einen nicht erfolgreichen Außenplanaritätstest, da der Graph nicht weiter abgebaut werden kann

12 Graphen mit beschränkter Baumweite

12.1 Definitionen und Einführung

Definition. Eine *Baumdekomposition* eines Graphen $G = (V_G, E_G)$ ist ein Paar $(\mathcal{X}, T)$, wobei $T = (V_T, E_T)$ ein Baum ist und $\mathcal{X} = \{X_u | u \in V_T\}$ eine Menge von Teilmengen $X_u \subset V_G$ (genau eine Menge X_u für jeden Knoten $u \in V_T$), so daß

1. $\bigcup_{u \in V_T} X_u = V_G$,
2. für jede Kante $\{u_1, u_2\} \in E_G$ gibt es einen Knoten $u \in V_T$ mit $u_1 \in X_u$ und $u_2 \in X_u$,
3. für jeden Knoten $w \in V_G$ ist der Teilgraph von T, der durch die Baumknoten $u \in V_T$ mit $w \in X_u$ induziert wird, zusammenhängend.

Die *Weite* der Baumdekomposition ist $\max_{u \in V_T} |X_u| - 1$.
Die *Baumweite* $\mathrm{BW}(G)$ eines Graphen G ist die minimale Weite aller Baumdekompositionen für G.

Beispiel. Betrachte Abbildung 12.1. Der Graph G mit Knotenmenge $\{1, \ldots, 6\}$ hat Baumweite 1. Seine Baumdekomposition ist $(\mathcal{X}, T)$, wobei T der Baum mit den Knoten $\{a, \ldots, e\}$ ist und $\mathcal{X} = \{X_a, \ldots, X_e\}$ mit $X_a = \{1, 2\}$, $X_b = \{2, 3\}$ usw.
Der Graph in Abbildung 12.2 hat eine Baumweite von 2.

Bemerkung.
1. Jeder Wald hat eine Baumweite von höchstens 1.
2. Jeder Graph mit einer Baumweite von höchstens 1 ist ein Wald.
3. Alle 2-reduzierbaren Graphen haben eine Baumweite von höchstens 2.

Definition. Eine Graphklasse C, also eine Menge von Graphen, hat eine *beschränkte Baumweite*, wenn es eine Zahl $k \in \mathbb{N}$ gibt, so daß alle Graphen aus

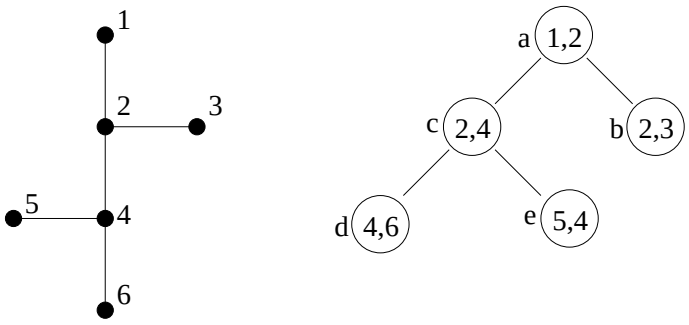


Abbildung 12.1: Beispiel für einen Graphen mit Baumweite 1

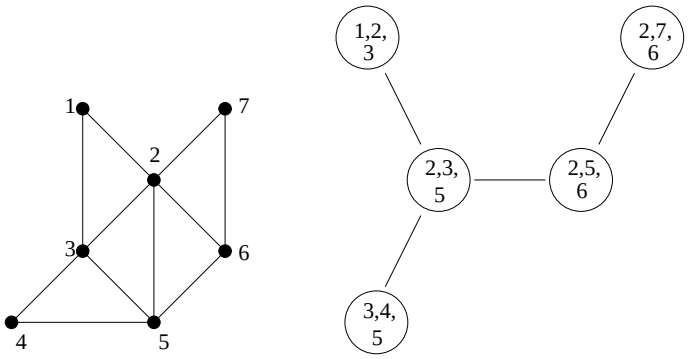


Abbildung 12.2: Beispiel für einen Graphen mit Baumweite 2

C eine Baumweite von höchstens k haben. Wenn k die kleinste Zahl mit obiger Eigenschaft ist, dann hat C die Baumweite k.

Bemerkung. Die Menge der vollständigen Graphen $C = \{K_n | n \geq 1\}$ hat keine beschränkte Baumweite, da der K_n die Baumweite $n-1$ hat. (Übungsaufgabe)

Satz 12.1. *Sei $G = (V, E)$ ein Graph.*

1. *Jeder Teilgraph $G' = (V', E')$ von G hat höchstens die Baumweite von G.*
2. *Die Baumweite von G ist das Maximum der Baumweiten der zusammenhängenden Komponenten von G.*
3. *Die Baumweite von G ist das Maximum der Baumweiten der zweifach Zusammenhangskomponenten von G.*

Beweis. 1. Sei $(\mathcal{X}, T)$ eine Baumdekomposition für G. Entferne alle Knoten der Menge $V \setminus V'$ aus den Mengen X_u mit $u \in V_T$. Das Ergebnis ist eine Baumdekomposition für G' mit gleicher oder kleinerer Weite.

2. Nach (1) ist die Baumweite jeder Zusammenhangskomponente von G höchstens die Baumweite von G. Seien $(\mathcal{X}_1, T_1), \ldots, (\mathcal{X}_r, T_r)$ Baumdekompositionen der r Zusammenhangskomponenten von G. Bilde die knotendisjunkte Vereinigung der Bäume $T_1, \ldots, T_r$ und füge $r-1$ neue Kanten hinzu, so daß ein Baum T enststeht. Die Menge $\mathcal{X}$ ist die disjunkte Vereinigung der $\mathcal{X}_i$.

3. Sei o.B.d.A. G zusammenhängend. Sei C die Menge der Artikulationspunkte für G (d.h.: bei Herausnahme eines Knotens aus C zerfällt G in mindestens 2 zusammenhängende Komponenten). Seien $(\mathcal{X}_1, T_1), \ldots, (\mathcal{X}_r, T_r)$ Baumdekompositionen der r zweifach Zusammenhangskomponenten für G. Sei T die disjunkte Vereinigung der Bäume $T_1, \ldots, T_r$ und $\mathcal{X}$ die disjunkte Vereinigung der Mengen $\mathcal{X}_i$, $1 \leq i \leq r$.

   Füge nun für jeden Artikulationspunkt $x \in C$ einen neuen Knoten u_x mit $X_{u_x} = \{x\}$ zu T hinzu. Füge für jede zweifach Zusammenhangskomponente

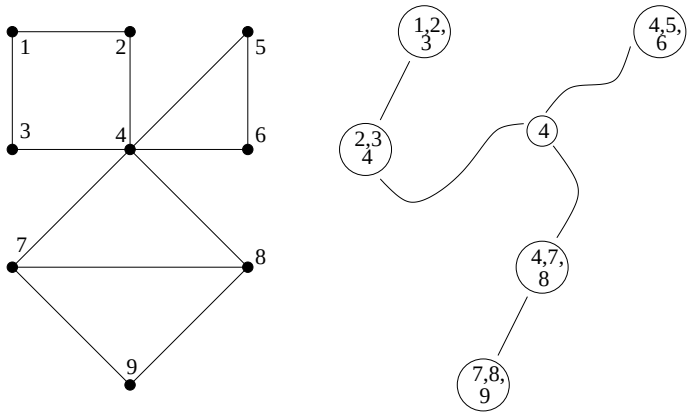


Abbildung 12.3: Zum Beweis von Satz 12.1, (3)

B_j, die x enthält, eine Kante von u_x zu einem Knoten v aus T_j mit $x \in X_v$ hinzu. Das Ergebnis ist eine Baumdekomposition für G.

Die Baumweite von $(\mathcal{X}, T)$ ist somit das Maximum der Baumweiten von $(\mathcal{X}_1, T_1), \ldots, (\mathcal{X}_r, T_r)$

□

Satz 12.2. *Sei $G = (V, E)$ ein Graph mit Baumweite k. Dann gibt es eine Baumdekomposition $(\mathcal{X}, T)$ für G mit einer Weite von k, so daß $|V_T| \in O(|V|)$ und T ein orientierter Baum ist (d.h.: ein Baum mit einer Wurzel, in dem jeder innere Knoten genau zwei Söhne hat).*

Beweis. Sei $(\mathcal{X}, T)$ eine Baumdekomposition für G. Wähle einen beliebigen Knoten $r \in V_T$ als Wurzel und wiederhole die folgenden Anweisungen:

- Entferne jedes Blatt $x \in V_T$ mit $x \neq r$ und $X_x \subseteq X_y$, wobei y der Vater von x in T ist.
- Für jeden Knoten $x \in V_T$ vom Grad 2 mit $x \neq r$ sei y der Vater von x und z der Sohn von x in T. Wenn $X_x \subseteq X_y$, dann entferne x und mache y zum neuen Vater von z.

Das Ergebnis ist weiterhin eine Baumdekomposition für G.
Wir zeigen: $|V_T| \in O(|V|)$.

1. Die Anzahl der Knoten mit 2 oder mehr Söhnen ist höchstens so groß wie die Anzahl der Blätter in T.

   Sei $x \in V_T$, $x \neq r$ ein Knoten mit genau einem Sohn und sei y der Vater von x. Da $X_x \neq X_y$, enthält X_x mindestens einen Knoten $u_x \in V$, der nicht in X_y ist. Für jedes Knotenpaar $x_1, x_2 \in V_T$, $r \neq x_1 \neq x_2 \neq r$ mit höchstens einem Sohn gilt: $u_{x_1} \neq u_{x_2}$. Ansonsten ist die dritte Eigenschaft der Baumdekomposition nicht gegeben. Also gibt es höchstens $|V|$ viele Knoten mit höchstens einem Sohn.

   Somit ist die gesamte Anzahl der Knoten aus $O(|V|)$.

2. Sei $x \in V_T$ ein innerer Knoten in T. Wenn x genau einen Sohn z hat, dann füge einen zweiten Sohn z' für x mit $X_{z'} = X_x$ hinzu.

   Wenn x die Söhne $z_1, \ldots, z_d$, $d > 2$ hat, dann ersetze x durch $d-1$ Knoten $x_1, \ldots, x_{d-1}$ mit $X_{x_i} = X - x$ für $1 \leq i \leq d-1$ (vgl. Abbildung 12.4).

   Der Vater von x_1 ist der Vater von x, die Söhne von x_i für $1 \leq i < d-1$ sind z_i und x_{i+1}.

   Die Söhne von X_{d-1} sind z_{d-1} und z_d.

   Dadurch werden höchstens $O(|V|)$ viele Knoten hinzugefügt.

□

Satz 12.3. *Ein Graph $G = (V, E)$ mit n Knoten und einer Baumweite von k hat höchstens*

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

Kanten.

Beweis. Induktion über n.

- Wenn $n < k+1$, dann ist die Baumweite von G kleiner als k.
- Wenn $n = k+1$ ist, dann hat $G = K_{n+1}$ genau $\frac{1}{2}k(k+1) = kn - \frac{1}{2}k(k+1)$ Kanten.
- Sei $n > k+1$ und $(\mathcal{X}, T)$ eine Baumdekomposition für G mit einer Weite von k, so daß für jeden Knoten $x \in V_T$ und für jeden Nachbarn $y \in V_T$ von x gilt: $X_x \not\subseteq X_y$.

  Sei $x \in V_T$ eine Knoten vom Grad 1 und $y \in V_T$ der Nachbar von x. Dann gibt es mindestens einen Knoten $u_x \in V$ mit $u_x \in X_x$ aber $u_x \notin X_y$. Da $|X_x| \leq 1k+1$, ist u_x mit höchstens k Knoten verbunden. Der Graph $G \backslash \{u_x\} =$

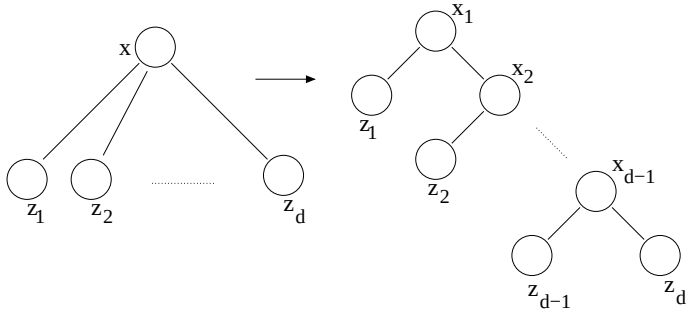


Abbildung 12.4: Zum Beweis von Satz 12.2

$\big(V \setminus \{u_x\}, E'\big)$ hat ebenfalls eine Baumweite von höchstens k. Somit gilt

$$\begin{aligned} |E| &\leq |E'| + k \\ &\leq k(n-1) - \frac{1}{2}k(k+1) + k \\ &\leq kn - \frac{1}{2}k(k+1) \end{aligned}$$

□

Folgerung. Jeder Graph mit einer Baumweite von höchstens 2 ist 2-reduzierbar.

12.2 Baumweite von planaren Gaphen

Definition. Sei $G = (V, E)$ ein Graph. Eine Knotenmenge $W \subseteq V$ ist ein *Knotenseparator* für G, wenn $G \setminus W$ nicht zusammenhängend ist.
Die Knotenmenge $W \subseteq V$ ist ein *balancierter Knotenseparator*, wenn die Knotenmenge V' von $G \setminus W$ in zwei Mengen V_1 und V_2 aufgeteilt werden kann, so daß folgendes gilt:

1. $V_1 \cup V_2 = V'$
2. $V_1 \cap V_2 = \emptyset$
3. Es gibt keine Kante $\{u, v\}$ in $G \setminus W$ mit $u \in V_1$, $v \in V_2$
4. Es gibt eine Zahl $k \in \mathbb{R}$ mit $k > 0$, so daß $|V_1| \geq \frac{|V'|}{k}$ und $|V_2| \geq \frac{|V'|}{k}$

Für $k = 3$ nennen wir W einen $\left(\frac{1}{3}, \frac{2}{3}\right)$-Knotenseparator.

Satz 12.4. *Jeder Baum hat einen $\left(\frac{1}{3}, \frac{2}{3}\right)$-Knotenseparator mit $|W| = 1$*

Beweis. Sei $T = (V, E)$ ein Baum. Wähle einen beliebigen Knoten $u \in V$. Wenn $T \setminus \{u\}$ einen Baum T' enthält mit mehr als $\frac{2}{3}\big(|V| - 1\big)$ Knoten, dann wähle als nächsten Knoten den Knoten $v \in T'$, der in T mit u adjazent ist.
Beobachtung 1: Der Auswahlprozess terminiert, da der Baum im Wald $T \setminus \{v\}$, der den Knoten u enthält, weniger als $\frac{2}{3}\big(|V| - 1\big)$ Knoten enthält.
Beobachtung 2: Wenn jeder Baum $T \setminus \{u\}$ in dem Wald $\leq \frac{2}{3}\big(|V| - 1\big)$ Knoten hat, dann können die Bäume in $T \setminus \{u\}$ so zu zwei Wäldern zusammengefasst werden, so daß jeder der beiden Wälder mindestens $\frac{1}{3}\big(|V| - 1\big)$ und höchstens $\frac{2}{3}\big(|V| - 1\big)$ Knoten enthält. □

Satz 12.5. *Jeder Graph mit einer Baumweite von k und mehr als $4(k+1)$ [1] Knoten hat einen $\left(\frac{1}{3}, \frac{2}{3}\right)$-Knotenseparator W mit $|W| \leq k + 1$.*

Beweis. Sei $G = (V, E)$ ein Graph und $(\mathcal{X}, T)$ eine Baumdekomposition für G der Weite k. Wähle einen Knoten $u \in V_T$ und betrachte die Bäume in $T \setminus \{u\}$.
Sei $T' = (V_{T'}, E_{T'})$ ein Baum in $T \setminus \{u\}$ und $W' = \Big(\bigcup_{v \in V_{T'}} X_v\Big) \setminus X_u$. Wenn W' mehr als $\frac{2}{3}\big(|V| - |X_u|\big)$ Knoten hat, dann wähle als nächsten Knoten den Knoten u' aus T', der mit u in T adjazent ist.
Beobachtung 1: Der Auswahlprozess terminiert (Übungsaufgabe).
Beobachtung 2: Der Graph G zerfällt nach Herausnahme der $|X_u|$ Knoten in zusammenhängende Komponenten mit höchstens $\frac{2}{3}\big(|V| - |X_u|$ Knoten. □

[1] Wie man auf diese Zahl kommt, ist eine Übungsaufgabe.

Definition. Eine Graphklasse C hat einen beschränkten balancierten Knotenseparator, wenn es ein $k \in \mathbb{N}$ gibt, so daß jeder Graph $G \in C$ einen balancierten Knotenseparator W mit höchstens k Knoten hat.

Bemerkung. Planare Graphen haben keinen beschränkten balancierten Knotenseparator und sind somit auch nicht baumweitebeschränkt.

Beispiel. Das Gitter mit n^2 Knoten ist planar. Zur Größe des Separators betrachte Kapitel 15, Separatoren und planare Graphen.

12.3 Wegweite

Definition. Eine *Wegdekomposition* eines Graphen $G = (V, E)$ ist eine Baumdekomposition $(\mathcal{X}, T)$ für G, wobei der Baum T ein Weg ist. (D.h.: Alle Knoten in T haben den Grad ≤ 2.) Die *Wegweite* $\mathrm{WW}(G)$ ist die minimale Weite aller Wegdekomposition für G.

Problem. Minimale Baumweite, minimale Wegweite
gegeben: Ein Graph G
gesucht: $BW(G)$, $WW(G)$

Bemerkung. Minimale Baumweite und minimale Wegweite sind NP-vollständig. Für jedes $k \in \mathbb{N}$ kann die Frage $BW = k$, $WW = k$ jedoch in linearer Zeit entscheiden werden. Eine Baumdekomposition (bzw. Wegdekomposition) kann ebenfalls in linearer Zeit konstruiert werden. Die Algorithmen sind jedoch sehr kompliziert und praktisch kaum brauchbar.

12.4 Algorithmen für Graphen mit beschränkter Baumweite

Definition. Sei $t \in \mathbb{N}_0$. Ein *t-terminaler Graph* ist ein Tripel $G = \big(V, E, (u_1, \ldots, u_t)\big)$, wobei (V, E) ein Graph und $u_1, \ldots, u_t \in V$ mit $u_i \neq u_j$ für $i \neq j$ eine Folge von t paarweise verschiedenen Knoten ist.

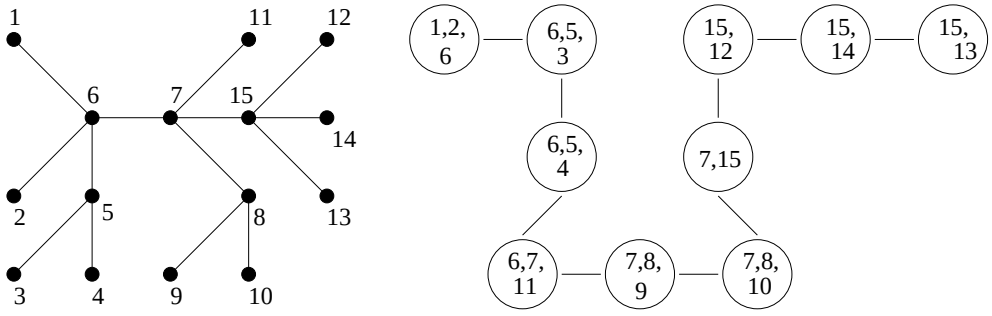


Abbildung 12.5: Ein Baum mit Wegweite 2

Seien $G_1 = \big(V_1, E_1, (u_1, \ldots, u_t)\big)$ und $G_2 = \big(V_2, E_2, (v_1, \ldots, v_t)\big)$ zwei t-terminale Graphen. Der Graph $G_1 \circ G_2$ ist wie folgt definiert:

- Bilde die disjunkte Vereinigung der Graphen (V_1, E_2), (V_2, E_2).
- Entferne den Knoten u_i und verbinde jeden Nachbarn von u_i mit v_i für $i = 1, \ldots, t$.

Sei $\mathcal{G}$ die Menge aller Graphen und $\mathcal{G}_t$ die Menge aller t-terminalen Graphen.

Definition. Sei $P : \mathcal{G} \to \{0, 1\}$ ein Entscheidungsproblem für Graphen, d.h. $p(G) = 1$ genau dann, wenn G die Eigenschaft P erfüllt.
Zwei t-terminale Graphen G_1 und G_2 sind *ersetzbar* bzgl. der Eigenschaft P (in Zeichen: $G_1 \sim_{t,P} G_2$), wenn für alle $H \in \mathcal{G}_t$ gilt:

$$P(G_1 \circ H) = P(G_2 \circ H).$$

Beispiel. $t = 3$, $P =$ Existenz eines Hamiltonkreises
Betrachte Abbildung 12.7. Die beiden 3-terminalen Graphen G_1 und G_2 sind nicht ersetzbar, weil es einen Graphen H gibt, so daß $G_1 \circ H$ keinen Hamiltonkreis besitzt, $G_2 \circ H$ besitzt einen, also $P(G_1 \circ H) \neq P(G_2 \circ H)$.

Bemerkung. $\sim_{t,P}$ ist eine Äquivalenzrelation. Für einen t-terminalen Graphen G sei $[G]_P$ die Menge aller $H \in \mathcal{G}_t$ mit $H \sim_{t,P} G$.

Definition. Eine Grapheigenschaft P heißt *recognizable*, wenn für jedes $t \geq 0$ die Relation $\sim_{t,p}$ einen endlichen Index hat (d.h.: es gibt endlich viele Äquivalenzklassen).

Beispiel. Für $t = 3$ und $P =$ Zusammenhang gibt es 6 Äquivalenzklassen (vgl. Abbildung 12.8).

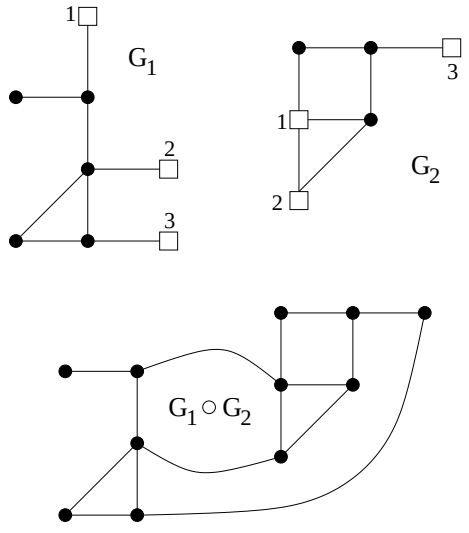


Abbildung 12.6: Zwei 3-terminale Graphen und ihre Komposition

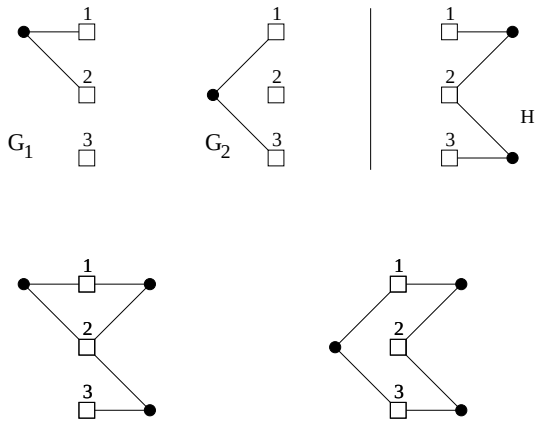


Abbildung 12.7: G_1 und G_2 sind nicht ersetzbar

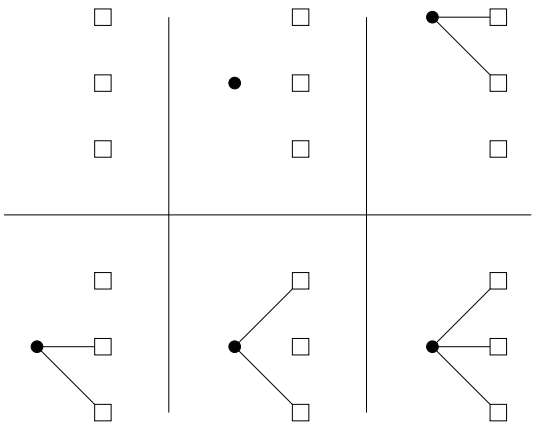


Abbildung 12.8: Die 6 Äquivalenzklassen für $P =$ Zusammenhang und $t = 3$

12.5 Algorithmus zur Bestimmung eines äquivalenten Repräsentanten

Sei P eine recognizable Eigenschaft und r_t die Anzahl der Äquivalenzklassen der Relation $\sim_{t,P}$. Seien $R_1, \ldots, R_{r_t} \in \mathcal{G}_t$ mit $R_i \not\sim_{t,P} R_j$ für $i \neq j$ (also genau ein Repräsentant aus jeder Klasse).

Lemma 12.1. *Zwei t-terminale Graphen $G_1, G_2 \in \mathcal{G}$ sind genau dann ersetzbar bezüglich P, ($G_1 \sim_{t,P} G_2$), wenn für alle $H \in \{R_1, \ldots, R_{r_t}\}$ gilt:*

$$P(G_1 \circ H) = P(G_2 \circ H)$$

Beweis. $\Rightarrow$: Wenn $G_1 \sim_{t,P} G_2$, dann gilt für alle $H \in \mathcal{G}_t$, daß $P(G_1 \circ H) = P(G_2 \circ H)$, also auch für $H \in \{R_1, \ldots, R_{r_t}\}$.

$\Leftarrow$: durch Widerspruch.

Angenommen, für alle $H \in \{R_1, \ldots, R_{r_t}\}$ gilt

$$P(G_1 \circ H) = P(G_2 \circ H),$$

aber es gibt ein $H' \in \mathcal{G}_t$ mit

$$P(G_1 \circ H') \neq P(G_2 \circ H').$$

Dann gibt es ein i mit $H' \in [R_i]$. Somit gilt

$$P(G_1 \circ R_i) \neq P(G_2 \circ R_i).$$

Das steht im Widerspruch zu unserer Annahme.

□

Definition. Sei $G = (V, E, Q) \in \mathcal{G}_t$ ein t-terminaler Graph und $Q_i = (v_{i,1}, \ldots, v_{i,t_i})$ für $i = 1, \ldots, k$ eine Folge von t_i paarweise verschiedenen Knoten aus V. Weiter sei $J_i \in \mathcal{G}_{t_i}$. Definiere den Graphen

$$G\big[Q_1/J_1, \ldots, Q_k/J_k\big] = (V', E', Q')$$

wie folgt (vgl. Abbildung 12.9):
Bilde die disjunkte Vereinigung der Graphen $G, J_1, \ldots, J_k$ und identifiziere den j-ten Knoten aus der Knotenliste von J_i mit dem j-ten Knoten aus Q_i für $1 \leq i \leq k$ und $1 \leq j \leq t_i$.
Die Knotenfolge Q' für $G\big[Q_1/J_1, \ldots, Q_k/J_k\big]$ ist die Knotenfolge Q von G.

Lemma 12.2. *Sei $G = (V, E, Q)$ ein t-terminaler Graph. Sei $Q_i = (v_{i,1}, \ldots, v_{i,t_i})$ für $i = 1, \ldots, k$ eine Folge von t_i paarweise verschiedenen Knoten aus V. Seien $J_i, J_i' \in \mathcal{G}_{t_i}$, so daß $J_i \sim_{t_i,P} J_i'$ für $i = 1, \ldots, k$ und eine Grapheigenschaft P. Dann gilt:*

$$G\big[Q_1/J_1, \ldots, Q_k/J_k\big] \sim_{t,P} G\big[Q_1/J_1', \ldots, Q_k/J_k'\big]$$

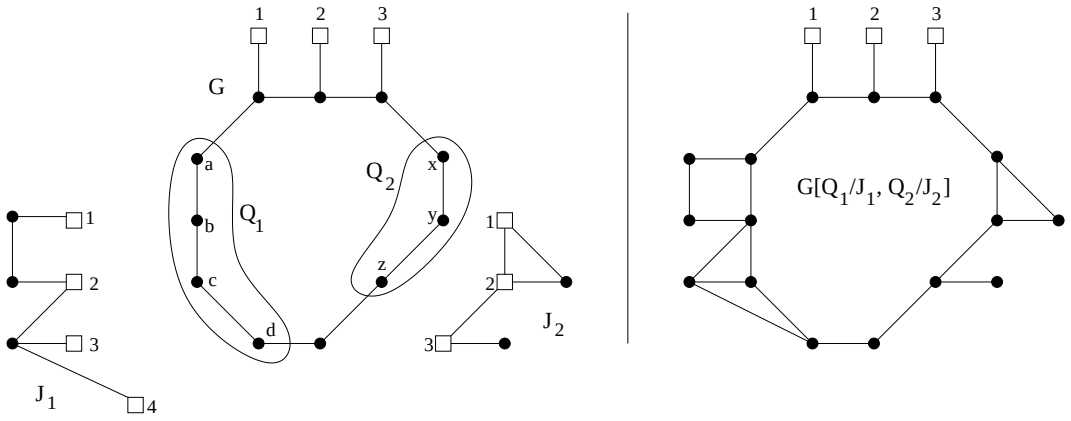


Abbildung 12.9: Zur Definition von $G\big[Q_1/J_1, \ldots, Q_k/J_k\big]$ (rechts)

Beweis. Induktion über k.

$k = 0$: $G \sim_{t,P} G$ folgt aus der Definition von $\sim_{t,P}$

$k > 0$: Sei $H \in \mathcal{G}_t$

$$
\begin{aligned}
&P\big(G[Q_1/J_1, \ldots, Q_K/J_k] \circ H\big) \\
&\quad = P\Big(\underbrace{\big(G[Q_2/J_2, \ldots, Q_k/J_k]\big)}_{\in \mathcal{G}_t}[Q_1/J_1] \circ H\Big) \text{ weil die Ersetzung assoziativ ist} \\
&\quad = P\Big(\big(G[Q_2/J_2, \ldots, Q_k/J_k]\big)[Q_1/J_1'] \circ H\Big) \text{ da } J_1 \sim_{t_1,P} J_1' \\
&\quad = P\Big(\big(G[Q_1/J_1']\big)[Q_2/J_2, \ldots, Q_k/J_k] \circ H\Big) \text{ weil die Ersetzung assoziativ ist} \\
&\quad = P\Big(\big(G[Q_1/J_1']\big)[Q_2/J_2', \ldots, Q_k/J_k'] \circ H\Big) \text{ nach Induktionsvoraussetzung} \\
&\quad = P\big(G[Q_1/J_1', \ldots, Q_K/J_k'] \circ H\big)
\end{aligned}
$$

□

Der Entscheidungsalgorithmus

Betrachte eine Baumdekomposition $(\mathcal{X}, T)$, so daß T ein binärer geordneter Baum ist [2]. Für jeden Knoten $u \in V_T$ sei $H_u = (V_u, E_u, Q_u) \in \mathcal{G}_t$ wie folgt definiert: (V_u, E_u) ist der Teilgraph von G, der durch die Knoten in den Mengen X_v induziert wird, wobei $v = u$ oder v ist Nachfolger von u in T. Sei u' der Vater von u. Die Knotenliste Q_u enthält die Knoten aus $X_u \cap X_{u'}$ in einer beliebigen aber festen Reihenfolge.
Berechne für jedes Blatt $u \in V_T$ die Klasse $[H_u]_P$.

[2] d.h.: T hat eine Wurzel, jeder Knoten außer die Wurzel hat genau einen Vater und die inneren Knoten haben genau zwei Söhne.

Sei u ein innerer Knoten in T mit den Söhnen v_1 und v_2. Berechne $[H_u]_P$ mit Hilfe der Klassen $[H_{v_1}]_P$ und $[H_{v_2}]_P$. Analysiere dazu einen t-terminalen Graphen $H'_u = G[Q_1/H'_{v_1}, Q_2/H'_{v_2}]$ mit $H'_{v_1} \in [H_{v_1}]_P$ und $H'_{v_2} \in [H_{v_2}]_P$ und G geeignet gewählt[3].
Das geht in Zeit $O(1)$.

Beispiel. Hamiltonkreis, wird noch nachgelieftert...

12.6 Monadische Logik 2.Ordnung für Graphen

Objekte: Knoten $u \in V$ und Kanten $e \in E$
Prädikate: Knotengleichheit: $u \stackrel{?}{=} v$
Kantengleichheit $e_1 \stackrel{?}{=} e_2$
Inzidenz $u \stackrel{?}{\in} e$
Verknüpfungen: $\neg$, $\vee$, $\wedge$

Quantifizierungen 1. Ordnung. Sei $W \subseteq V$, $F \subseteq E$

$$\forall u \in W, \quad \exists u \in W, \quad \forall e \in F, \quad \exists e \in F$$

Quantifizierungen 2. Ordnung. Sei $W \subseteq V$, $F \subseteq E$

$$\forall W' \subseteq W^k, \quad \exists W' \subseteq W^k, \quad \forall F' \subseteq F^k, \quad \exists F' \subseteq F^k$$

Monadische Quantifizierung 2. Ordnung. $k = 1$
Aufbau weiterer Prädikate.

- **Adjazenz.** Knoten u_1 und u_2 sind adjazent $\big(\mathrm{adj}(u_1, u_2)\big)$ genau dann, wenn

$$\exists e \in E \,:\, u_1 \in e \wedge u_2 \in e$$

- **Mengenzugehörigkeit.**

$$u \in W \quad \Leftrightarrow \quad \exists u' \in W \,:\, u = u'$$

- **Knotengrad $\geq k$.**

$$\mathrm{grad}(u) \geq k \;\Leftrightarrow\; \exists u_1, \ldots, u_k \in V,\; u_1 \neq u_2, u_1 \neq u_3, \ldots, u_1 \neq u_k, u_2 \neq u_3, \ldots u_{k-1} \neq u_k \\ \wedge\, \mathrm{adj}(u, u_1) \wedge \ldots \wedge \mathrm{adj}(u, u_k)$$

[3] G ist im wesentlichen der Teilgraph, der von den Knoten in X_u induziert wird. Allerdings muß man die Entstehung von Mehrfachkanten vermeiden.

- **Knotengrad $= k$.**

$$\text{grad}(u) = k \quad \Leftrightarrow \quad \text{grad}(u) \geq k \wedge \neg\, \text{grad}(u) \geq k+1$$

- **Alle Knoten haben Grad $= 2$.**

$$\forall u \in V \,:\, \text{grad}(u) = 2$$

- G **ist zusammenhängend.**

$\forall\, V' \subseteq V$ mit $V' \neq \emptyset$ und $V' \neq V : \exists u_1 \in V'$ und $\exists u_2 \in V \backslash V'$ mit $\text{adj}(u_1, u_2)$

- G **ist ein Kreis.** G ist zusammenhängend und alle Knoten haben Grad 2
- G **enthält einen Hamiltonkreis.**

$$\Leftrightarrow \exists E' \subseteq E : (V, E') \text{ ist ein Kreis}$$

- G **ist dreifärbbar.** [4]

$$\begin{aligned} \exists V_1, V_2, V_3 \subseteq V \text{ mit} \quad & V_1 \cup V_2 \cup V_3 = V, \\ \wedge \;\; & V_1 \cap V_2 = \emptyset, V_1 \cap V_3 = \emptyset, V_2 \cap V_3 = \emptyset, \\ \wedge \;\; & \forall u_1, u_2 \in V_1 \,:\, \neg\, \text{adj}(u_1, u_2) \\ \wedge \;\; & \forall u_1, u_2 \in V_2 \,:\, \neg\, \text{adj}(u_1, u_2) \\ \wedge \;\; & \forall u_1, u_2 \in V_3 \,:\, \neg\, \text{adj}(u_1, u_2) \end{aligned}$$

Satz 12.6 (Courcelle)**.** *Jede Grapheigenschaft P, die in monadischer Logik 2.Ordnung mit Quantifizierungen über Knoten und Kanten sowie Knoten- und Kantenmengen definierbar ist, ist recognizable.*

Nicht definierbar sind Ordnungen und Größenvergleiche wie z.B. $|V_1| \stackrel{?}{=} |V_2|$.
Optimierungsprobleme können auf Graphen mit beschränkter Baumweite oft mit zusätzlichen Informationen gelöst werden.

12.7 Partielle k-Bäume

Definition. Der vollständige Graph mit k Knoten ist ein k-Baum.
Sei G ein k-Baum. Fügt man einen Knoten zu G hinzu und verbindet man u mit k Knoten $u_1, \ldots u_k$, die alle paarweise adjazent sind, so erhält man wieder einen k-Baum. Ein *partieller k-Baum* ist ein Teilgraph eines k-Baumes.

[4] Die hier verwendeten Operationen $\cap$ und $\cup$ kann man ebenfalls in monadischer Logik 2.Ordnung ausdrücken.

Satz 12.7. *Die Menge aller partiellen k-Bäume ist genau die Menge aller Graphen mit einer Baumweite von $\leq k$.*

Beobachtung. Wenn G ein Graph ist, der einen vollständigen Teilgraphen K_n mit $n \geq 2$ enthält, dann gibt es in jeder Baumdekomposition $(\mathcal{X}, T)$ für G einen Knoten $u \in V_T$, so daß alle Knoten aus K_n in X_u enthalten sind.

Beweis von Satz 12.7. $\Rightarrow$: Der vollständige Teilgraph K_{k+1} hat eine Baumweite von k. Jeder Teilgraph von K_{k+1} hat also höchstens eine Baumweite von k.

Sei G ein k-Baum und $(\mathcal{X}, T)$ eine Baumdekomposition für G mit einer Weite $\leq k$. Sei G' der Graph, der durch Hinzufügen eines Knotens u entstanden ist..

Erweitere T um einen Knoten v und verbinde v mit einem Knoten $w \in V_T$, so daß X_w alle Nachbarn $u_1, \ldots, u_k$ enthält. Erweitere $\mathcal{X}$ um $X_v = \{u, u_1, \ldots, u_k\}$.

$\Rightarrow \mathrm{BW}(G') \leq k$

$\Leftarrow$: Induktion über die Knotenanzahl.

Jeder Graph mit $\leq k+1$ Knoten und einer Baumweite $\leq k$ ist ein partieller k-Baum.

Sei G ein Graph und $(\mathcal{X}, T)$ eine Baumdekomposition mit einer Weite $\leq k$. Verteile die Knoten in den Mengen $X_u \in \mathcal{X}$ so, daß $|X_u| = k+1$ für alle $u \in V_T$.

Sei $u \in V_T$ ein Blatt in T. Verbinde alle Knoten in X_u paarweise mit Kanten. Sei $w \in X_u$ mit $w \notin X_{u'}$, wobei u' mit u in T adjazent ist. Der Graph G ohne w ist nach Induktion ein partieller k-Baum. Da w mit einem vollständigem Teilgraphen mit k Knoten verbunden ist, ist G ebenfalls ein partieller k-Baum.

□

13 Chordale Graphen

13.1 Definitionen und Einführung

Chordale Graphen sind unter verschiedenen Namen bekannt, z.B.: triangulierte Graphen oder perfekte Eliminationsgraphen.

Definition. Sei C ein Kreis. Ein *Chord* für C ist eine Kante zwischen zwei nicht benachbarten Knoten. Ein Graph G ist *chordal*, wenn jeder Kreis in G mit einer Länge von ≥ 4 mindestens einen Chord hat.

Satz 13.1 (Dirac)**.** *Ein Graph G ist chordal, wenn jede minimale Separationsmenge $W \subseteq V$ für G einen vollständigen Teilgraphen induziert.*

Definition. Sei $G = (V, E)$ ein Graph. Für einen Knoten $u \in V$ sei $N(u)$ die Menge aller mit u adjazenten Knoten. Ein Knoten $u \in V$ heißt *simplizial* in G, wenn $N(u)$ eine Clique impliziert.
Eine Anordnung $(u_1, \ldots, u_n)$ der Knoten aus V ist eine *perfekte Eliminationsordnung* für G, wenn für alle $i \in \{1, \ldots, n\}$ der Knoten u_i simplizial in $G\big(\{u_i, \ldots, u_n\}\big)$ ist.

Satz 13.2 (Dirac, Fulkerson, Gross, Rose)**.** *Ein Graph G ist genau dann chordal, wenn er eine perfekte Eliminationsordnung besitzt.*

Weitere Knotenanordnungen

Seien $s_1 = (a_1, \ldots, a_k)$ und $s_2 = (b_1, \ldots, b_l)$ zwei Vektoren mit $a_i, b_i \in \mathbb{N}$. Wir sagen: s_1 ist *lexikographisch kleiner* als s_2 (in Zeichen: $s_1 < s_2$), wenn gilt:

1. Es gibt einen Index $i \leq \min\{l, k\}$, so daß $a_i < b_i$ und $a_j = b_j$ füt $j = 1, \ldots, i-1$ oder
2. $k < l$ und $a_i = b_i$ für $i = 1, \ldots, k$.

Für einen Vektor $s = (a_1, \ldots, a_k)$ und ein Element $a \in \mathbb{N}$ sei $s \circ a = (a_1, \ldots a_k, a)$.

13.2 Lexikographische Breitensuche, LexBFS

gegeben: Ein Graph $G = (V, E)$
gesucht: LexBFS-Ordnung σ (wird durch den Algorithmus definiert)
`for all` $v \in V$ `do` $l(v) = ()$;
`for` $n = |V|$ `downto` 1 `do`
{ `wähle einen Knoten` $v \in V$ `mit lexikographisch größtem Label` $l(v)$;
  $\sigma(n) = v$;
  `for all` $u \in V \cap N(v)$ `do` $l(u) = l(u) \circ n$;
  $V = V \setminus \{v\}$
}

Beispiel. Wendet man bei dem Graphen in Abbildung 13.1 den Algorithmus an (Startknoten ist a), so erhält man folgende Ordnung:

$$\sigma = (h, j, i, g, f, e, c, d, b, a)$$

Wird in dem Fall, daß mehrere Knoten das gleiche lexikographisch größte Label haben, ein Knoten mit größtem Grad gewählt, dann erhält man *Cardinality-LexBFS*.
Wird anstelle des lexikographisch größtem Knoten ein Knoten gewählt, der mit den meisten bereits besuchten Knoten adjazent ist, dann erhält man *Maximum-Cardinality-Search (MCS)*

Der MCS-Algorithmus
gegeben: Ein Graph $G = (V, E)$
gesucht: Eine MCS-Ordnung σ der Knoten in V

`for` $n = |V|$ `downto` 1 `do`
{ `wähle ein` $u \in V$ `mit einer maximalen Anzahl von bereits numerierten Nachbarn;`
  `numeriere` u `mit` n;

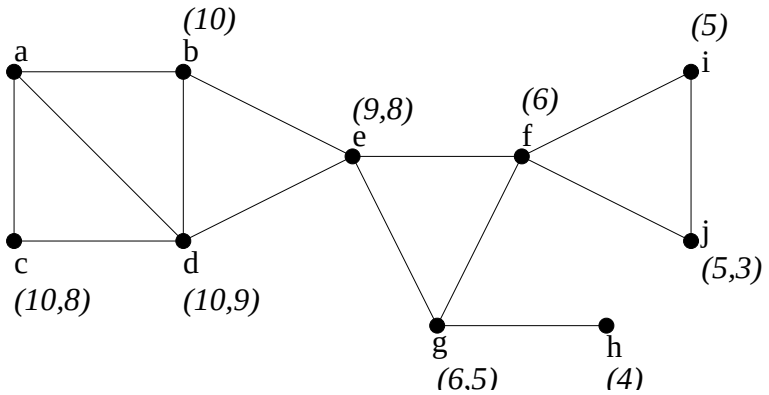


Abbildung 13.1: Ein chordaler Graph mit den Werten für l

$\sigma(n) = u$;
$V = V \setminus \{u\}$;
}

Bemerkung. Die LexBFS- und MCS-Ordnung können in linearer Zeit berechnet werden. (Übungsaufgabe).

Satz 13.3. *Sei $G = (V, E)$ ein Graph. Die folgenden Eigenschaften sind äquivalent:*

1. *G ist chordal.*
2. *Jede LexBFS-Ordnung für G ist eine perfekte Eliminationsordnung für G.*
3. *Jede MCS-Ordnung für G ist eine perfekte Eliminationsordnung für G.*

Ein einfacher Test auf Chordalität

Definition. Sei $G = (V, E)$ ein Graph und $\sigma = (u_1, \ldots, u_n)$ eine Knotenordnung für G. Der *Fill-In* $F(\sigma)$ ist die Menge aller Kanten $\{u_i, u_j\}$, $i \neq j$ mit folgenden Eigenschaften:

1. $\{u_i, u_j\} \notin E$
2. Es gibt einen Weg zwischen u_i und u_j, der nur Knoten u_l mit $l < i$ und $l < j$ enthält.

Satz 13.4. *Ein Graph G ist genau dann chordal, wenn es eine Knotenordnung σ gibt mit $F(\sigma) = \emptyset$.*

Der Algorithmus

1. Berechne eine Knotenfolge σ für G mit $F(\sigma) = \emptyset \Leftrightarrow G$ ist chordal.
2. Berechne $F(\sigma)$.

Satz 13.5 (Tarjan,Yannakakis)**.** *Sei G ein Graph und σ eine LexBFS- oder MCS-Ordnung für G. Dann gilt: G ist chordal $\Leftrightarrow F(\sigma) = \emptyset$.*

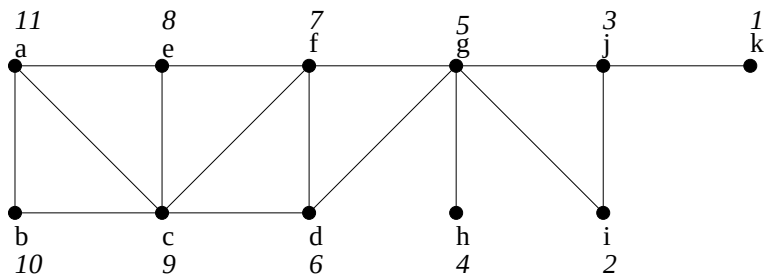


Abbildung 13.2: Ein chordaler Graph mit MCS-Numerierung der Knoten

Beispiel. Betrachte den Graphen in Abbildung 13.3. Die angegebene MCS-Ordnung $\sigma = (e, d, f, c, g, b, a)$ der Knoten hat ein nicht-leeres Fill-In (gestrichelte Kanten). Also ist dieser Graph nicht chordal.

Alternative Definition für chordale Graphen

Definition. Der Graph, der aus nur einem Knoten besteht, ist ein chordaler Graph.
Sei G ein chordaler Graph. Fügt man einen neuen Knoten u zu G hinzu und verbindet man u mit Knoten $u_1, \ldots, u_n$, die einen vollständigen Teilgraphen induzieren, so erhält man wieder einen chordalen Graphen.

Bemerkung. Die perfekte Eliminationsordnung kann oft dazu genutzt werden, um Graphenprobleme auf chordalen Graphen effizient zu lösen.

Beispiel. Maximale Clique
Durchlaufe die Knoten in der Reihenfolge der perfekten Eliminationsordnung $(u_1, \ldots, u_n)$ und berechne für u_i die Zahl $|N(u_i) \cap \{u_i, \ldots, u_n\}| + 1$. Das Maximum dieser Zahlen ist die Größe der maximalen Clique in G.

Definition. Sei $G = (V, E)$ ein Graph.

1. $\alpha(G) = \max\left\{|V'| \middle| V' \subseteq V; V \text{ ist eine unabhängige Menge}\right\}$
2. $\omega(G) = \max\left\{|V'| \middle| V' \subseteq V; V \text{ ist eine Clique}\right\}$
3. $\chi(G) = \min\left\{k \middle| \text{ es gibt eine Partition von } V \text{ in } k \text{ disjunkte unabhängige Mengen}\right\}$
4. $\kappa(G) = \min\left\{k \middle| \text{ es gibt eine Partition von } V \text{ in } k \text{ disjunkte Cliquen}\right\}$

Bemerkung. $\omega(G) \leq \chi(G)$ und $\alpha(G) \leq \kappa(G)$. $\chi(G)$ wird die *chromatische Zahl* von G genannt. Es gilt

$$\alpha(G) = \omega(\overline{G}) \text{ und } \chi(G) = \kappa(\overline{G})$$

Das Berechnen der Zahlen $\alpha(G), \omega(G), \chi(G), \kappa(G)$ ist für allgemeine Graphen NP-vollständig.

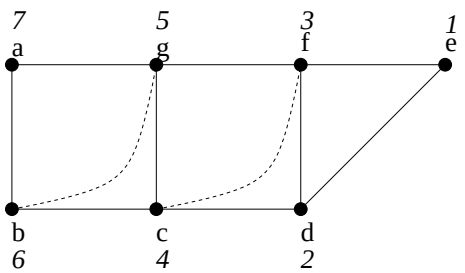


Abbildung 13.3: Ein Graph mit MCS-Sortierung σ und Fill-In $F(\sigma) = \{\{b, g\}, \{c, f\}\}$

Definition. Ein Graph G ist perfekt, wenn für jeden induzierten Teilgraphen H von G gilt: $\chi(H) = \omega(H)$.

Satz 13.6. *Wenn G perfekt ist, dann ist $\overline{G}$ perfekt.*

$$\chi(G) = \omega(G) \Leftrightarrow \chi(\overline{G}) = \omega(\overline{G})$$

Satz 13.7. *Jeder chordale Graph ist perfekt.*

Beweis. Berechne $\chi(G)$ und $\omega(G)$ mit Hilfe einer perfekten Eliminationsordnung $(u_1, \ldots, u_n)$.

$$\chi(G) = \omega(G) = 1 + \max_{i \in \{1,\ldots,n-1\}} \left| N(u_i) \cap \{u_i, \ldots, u_n\} \right|$$

□

13.3 Intervallgraphen

Definition. Sei $I = (I_1, \ldots, I_n)$ eine endliche Menge von Intervallen auf einer Linie. (Man kann o.B.d.A. annehmen, daß alle Anfangs- und Endpunkte der Intervalle aus $\mathbb{N}$ sind.)
Der *Intervallgraph* $G_I = (V, E)$ hat einen Knoten u_i für jedes Intervall I_i. Zwei Knoten u_i und u_j sind genau dann adjazent, wenn sich die Intervalle I_i und I_j überschneiden.

Satz 13.8. *Jeder Intervallgraph ist chordal.*

Beweis. Ordne die Intervalle $I_i = (a_{i,1}, a_{i,2})$ in I nach ihren Endpunkten. Dann ist die Knotenordnung $(u_1, \ldots, u_n)$ eine perfekte Eliminationsordnung. Knoten u_1 ist simplizial in G und nach Herausnahme von I_1 ist Knoten u_2 simplizial in $G \setminus \{u_1\}$. □

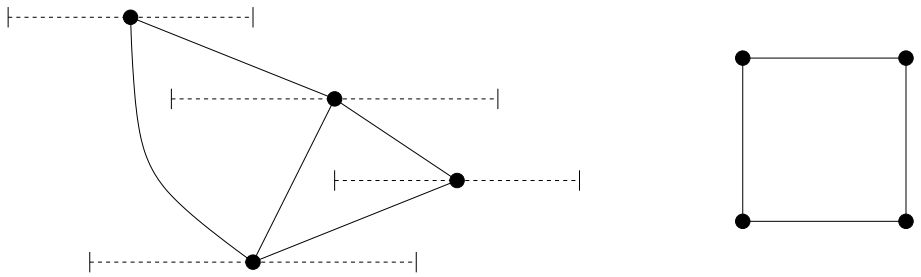


Abbildung 13.4: Vier Intervalle und der zugehörige Intervallgraph (links) und der kleinste Nicht-Intervallgraph (rechts)

Definition. Sei $G = (V, E)$ ein ungerichteter graph. Der gerichtete Graph $G' = (V, E')$ ist eine *Orientierung* für G, wenn für alle Kanten $\{u, v\} \in E$ entweder $(u, v) \in E'$ oder $(v, u) \in E'$ und für alle $(u, v) \in E'$ ist $\{u, v\} \in E$.
G' ist eine *transitive Orientierung* für G, wenn E' transitiv ist, d.h. sind $(u, v) \in E'$ und $(v, w) \in E'$, dann ist auch $(u, w) \in E'$.

Beobachtung 1. Ist $G = (V, E)$ ein Intervallgraph, dann ist für jede Teilmenge $W \subseteq V$ der durch W induzierte Teilgraph von G wieder ein Intervallgraph.
Beobachtung 2. Wenn G ein Intervallgraph ist, dann hat G keinen induzierten C_4[5], und $\overline{G}$ ist transitiv orientierbar.
Denn: Sind I_x und I_y zwei nicht überlappende Intervalle mit $a_{x,1} < a_{y,1}$ und $a_{x,2} < a_{y,2}$, dann orientiere die Kante $\{u_x, u_y\} \in \overline{E}$ von u_x nach u_y. Wenn es zwei Kanten $(u_x, u_y) \in \overline{E}'$ und $(u_y, u_z) \in \overline{E}'$ gibt, dann existiert auch die Kante $(u_x, u_z) \in \overline{E}'$. [6]
Beobachtung 3. Die maximalen Cliquen von G können so angeordnet werden, daß für jeden Knoten u die Cliquen, die u enthalten, nebeneinander stehen.

Satz 13.9 (Gilmore, Hoffman)**.** *Die folgenden Aussagen sind äquivalent:*

1. *G ist ein Intervallgraph.*
2. *G enthält keinen induzierten C_4 und $\overline{G}$ ist transitiv orientierbar.*
3. *Die maximalen Cliquen in G können so angeordnet werden, daß für jeden Knoten u die Cliquen, die u enthalten, nebeneinander stehen.*

Bemerkung. Intervallgraphen können in linearer Zeit (d.h.: in Zeit $O\big(|V|+|E|\big)$) erkannt werden.
Korte, Möhrin benutzen PQ-Bäume (1989)
Habib, Paul, Viennot benutzen LexBFS (1996)
Corneil, Olariu, Stewart benutzen 4-sweep LexBFS (1998)

Definition. Ein Intervallgraph G_I ist ein *eigener Intervallgraph*[7], wenn alle Intervalle sich nicht gegenseitig echt enthalten:

$$\forall I_x, I_y \in I \text{ gilt } I_x \not\subset I_y \text{ und } I_y \not\subset I_x$$

Ein Intervallgraph G_I ist ein *Einheitsintervallgraph*, wenn alle Intervalle in I die gleiche Länge haben.

Satz 13.10. *G ist genau dann ein eigener Intervallgraph, wenn er ein Einheitsintervallgraph ist.*

[5] C_n bezeichnet den Kreis mit n Knoten

[6] Dabei ist $\overline{E}$ die Kantenmenge des ungerichteten Komplementgraphen und $\overline{E}'$ die entsprechende gerichtete Kantenmenge.

[7] engl.: proper intervalgraph

Beweis($\Leftarrow$): Wenn alle Intervalle gleich groß sind, kann es keine echten Inklusionen geben.

($\Rightarrow$): Zeige eine schärfere Aussage: Sei $I = \{I_1, \ldots, I_k\}$ eine Menge von Intervallen, die sich nicht paarweise gegenseitig echt enthalten und sei o.B.d.A. $a_{i,1} < a_{i+1,1}$ für $i = 1, \ldots, k-1$. (Gleiche Intervalle werden gesondert betrachtet, daher echt kleiner.) Dann gilt offensichtlich auch $a_{i,2} < a_{i+1,2}$ für $i = 1, \ldots, k-1$.

Die Anfangs- und Endpunkte der Intervalle in I können so verändert werden, daß die resultierenden Intervallgraphen gleich sind, alle Intervalle die gleiche Länge haben und die Ordnung der Intervalle über die Anfangs- bzw. Endpunkte ebenfalls gleich bleibt.

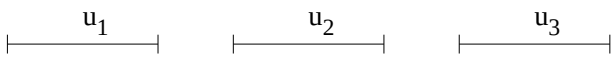
Induktion über die Anzahl der Intervalle: Verändere die Intervalle $I_1, \ldots I_{k-1}$ entsprechend der Induktionsvoraussetzung (für ein Intervall ist die Aussage trivial). Lege dann den Anfangspunkt für das Intervall I_k fest. Da alle Intervalle $I_1, \ldots I_k$ die gleiche Länge haben und vor I_k enden, ist der Anfangspunkt größer als alle übrigen Anfangspunkte.

□

Definition. Ein *asteroides Tripel* (AT) in einem Graphen $G = (V, E)$ ist eine unabhängige Menge von drei Knoten $\{u_1, u_2, u_3\}$, zu der es Wege $P_{1,1}, P_{1,3}, P_{2,3}$ gibt, so daß $P_{i,j}$ ein Weg von u_i nach u_j ist und kein Knoten auf dem Weg $P_{i,j}$ ist mit u_k adjazent für $\{i, j, k\} = \{1, 2, 3\}$.

Lemma 13.1. *Intervallgraphen sind AT-frei.*

Beweis. Jeder Weg $P_{1,3}$ im folgenden Bild enthält Nachbarn von u_2:



□

Satz 13.11. *Die Intervallgraphen sind genau die AT-freien chordalen Graphen.*

14 Bipartite Graphen

Definition. Ein Graph $G = (V, E)$ ist bipartit, wenn V disjunkt in zwei Knotenmengen V_1, V_2 zerlegt werden kann, so daß für alle $\{u, v\} \in E$ gilt: ($u \in V_1$ und $v \in V_2$) oder ($u \in V_2$ und $v \in V_1$).

Da jeder Teilgraph eines bipartiten Graphen wieder bipartit ist, sind bipartite Grpahen perfekte Graphen. Es gilt:

$$\begin{aligned} \chi(G) = \omega(G) &= 2 \qquad \text{falls } G \text{ mindestens eine Kante hat} \\ \chi(G) = \omega(G) &= 1 \qquad \text{sonst} \end{aligned}$$

Sei $G = (V; E)$ ein Graph mit $|V| = n$. Eine Menge ist genau dann unabhängig in $\overline{G}$, wenn sie eine Clique in G ist. Eine Partition von $\overline{G}$ in unabhängige Mengen entspricht einer Cliquenüberdeckung für G. Ist G bipartit, dann sind die Cliquen in G entweder Kanten oder einzelne Knoten. Eine minimale Cliquenüberdeckung für G ist ein Matching M mit den übrigen nicht am Matching beteiligten Knoten. Die Cliquenüberdeckung hat in diesem Fall die Größe $n - |M|$. Daher gilt:

$$\chi(\overline{G}) = \kappa(G) = n - \max_{M \text{ ist Matching für } G} |M|$$

Offensichtlich gilt auch:

$$\omega(\overline{G}) = \alpha(G) = n - \min_{A\text{ist Vertex-Cover für } G} |A|$$

Die Gleichheit $\chi(\overline{G}) = \omega(\overline{G})$ zeigt der folgende Satz:

Satz 14.1. *Sei $G = (V, E)$ ein bipartiter Graph. Die Größe eines maximalen Matchings entspricht der Größe eines minimalen Vertex-Covers.*

Beweis. Für jedes Vertex-Cover U und jedes Matching M gilt $|M| \leq |U|$, weil jedes Vertex-Cover U mindestens einen Knoten von jeder Kante in einem Matching M für G enthalten muß.
Sei $\tau(G) = \min\big\{|U| \;\big|\; U \text{ ist Vertex-Cover in } G\big\}$. Entferne so viele Kanten wie möglich aus G ohne $\tau(G)$ zu ändern. Für den so erzeugten Teilgraphen G' von G gilt folgendes:

1. $\tau(G) = \tau(G')$
2. $\forall e \in E_{G'}$ gilt $\tau\big(G' \setminus \{e\}\big) < \tau(G')$ und sogar $\tau\big(G' \setminus \{e\}\big) = \tau(G') - 1$

Behauptung: G' ist ein Matching für G.
Angenommen G' ist kein Matching für G, dann gibt es zwei Kanten $e_1 = \{u, v\}$ und $e_2\{v, w\}$ mit einem gemeinsamen Knoten $v \in G'$. Sei U_i , $i \in \{1, 2\}$ ein Vertex

Cover für $G \setminus \{e_i\}$ mit $|U_i| = \tau(G') - 1$. Da U_i kein Vertex Cover für G' ist, gilt $U_I \cap e_i = \emptyset$. betrachte nun den durch $\{v\} \cup (U_1 \Delta U_2)$ [8] induzierten Teilgraphen G'' von G'. Sei $t = |U_1 \cap U_2|$, dann hat G'' genau $1+2\big(\tau(G)-1\big)-2t = 1+2\big(\tau(G)-1-t\big)$ Knoten.
G'' ist bipartit, weil G bipartit ist.
sei T die kleinere der beiden Knotenmenge einer Bipartition für G''.
Behauptung: $S = T \cup (U_1 \cap U_2)$ ist ein Vertex Cover für G'.
Sei e eine Kante von G' mit $e \neq e_1, e_2$, dann wird e von U_1 und U_2 überdeckt.

1. Enthält U_1 und U_2 den gleichen Knoten von e, dann ist er auch in S.
2. Enthält U_1 einen Knoten u' und U_2 den anderen Knoten u'' von $e = \{u', u''\}$, dann ist e in G'' und somit ist entweder u' oder u'' in T und auch in S.

Da $|U_1 \cap U_2| = t$ und $|T| = \dfrac{2\big(\tau(G) - 1 - t\big)}{2} = \tau(G) - 1 - t$ erhalten wir den Widerspruch $|S| < \tau(G) - 1$. Also ist G' ein Matching der Größe $\tau(G)$. □

Lemma 14.1. *Jeder bipartite Graph $G = (V, E)$ ist transitiv orientierbar.*

Beweis. Sei V_1, V_2 eine Bipartition für G, dann orientiere die Kanten $\{u, v\} \in E$ mit $u \in V_1$, $v \in V_2$ von u nach v. □

Bemerkung. Eine transitive Orientierung eines transitiv orientierbaren Graphen kann in linearer Zeit konstruiert werden $O((n + m))$.

Korollar. Transitiv orientierbare Graphen sind perfekt. (Übungsaufgabe)
Sei $G = (V, E)$ ein transitiver orientierbarer Graph. Die Größe der größten Clique in G entspricht der Länge einer längsten Weges in einer transitiven Ordnung G' für G. Die Größe der größten unabhängigen Menge in G entspricht dr Größe einer maximalen Pfadzerlegung für G'. Beide Probleme können für transitiv orientierbare Graphen effizient gelöst werden.

Definition. Eine Pfadzerlegung für einen gerichteten Graphen $G = (V, E)$ ist eine Menge von paarweise knotendisjunkten Wegen in G, so daß jeder Knoten in genau einem Weg vorkommt.

Das Optimierungsproblem Weighted Path Partition

gegeben: ein gerichteter Graph $G = (V, E, f)$ mit einer Kantengewichtsfunktion $f : E \to \mathbb{R}$

gesucht: eine Pfadzerlegung $\mathcal{P} = \{P_1, \ldots, P_n\}$ mit maximalen Gewicht $\sum_{P_i \in \mathcal{P}} f(P_i)$, wobei $f(P_i)$ die Summe der Kantengewichte $f(e)$ der Kanten e in P_i ist

[8] $U_1 \Delta U_2$ ist die symmetrische Differenz der beiden Mengen

Weighted Path Partition ist NP-vollständig (Hamiltonweg). In ungerichteten Graphen ($f(e) = 1\ \forall e$) ist eine Pfadzerlegung mit möglichst vielen Kanten eine Pfadzerlegung mit möglichst wenig Pfaden.

$$|E(\mathcal{P})| = n - |\mathcal{P}|$$

Lemma 14.2. *In kreisfreien gerichteten Graphen reduziert sich das Problem Weighted Path Partition auf Maximum Weighted Matching für bipartite (ungerichtete) Graphen.*

Beweis. Sei $G = (V_G, E_G, f)$ mit $V_G = \{u_1, \ldots, u_n\}$ ein kreisfreier gerichteter Graph mit Kantengeichtsfunktion $f : E_G \to \mathbb{R}$. Der ungerichtete Graph $H = (V_H, E_H, h)$ mit Kantengewichtsfunktion $h : E_H \to \mathbb{R}$ ist wie folgt definiert:

$$\begin{aligned} V_H &= \{x_1, \ldots, x_n, y_1, \ldots, y_n\} \\ E_H &= \big\{\{x_i, y_j\} \;\big|\; (u_i, u_j) \in E_G\big\} \\ h\big(\{x_i, y_i\}\big) &= f\big((u_i, u_j)\big) \end{aligned}$$

Ein Matching in H entspricht einer Wegzerlegung für G. □

Sei $\nu(G) = \max\big\{|M| \;\big|\; M \text{ ist Matching für } G\big\}$. Wir wissen bereits: Für bipartite Graphen gilt: $\tau(G) = \nu(G)$. Sei G ein transitiv orientierbarer Graph und G' eine transitive Orientierung für G. Sei H wie oben definiert für den gerichteten Graphen G'. Also ist $\kappa(G) = n - \nu(H)$.

Satz 14.2. *Das Optimierungsproblem Independent Set (d.h. die Bestimmung von $\alpha(G)$) reduziert sich für transitiv orientierbare Graphen auf das bipartite Matching Problem.*

Beweis. Sei $H = (V_H, E_H)$ der bipartite Graph zur transitiven Orientierung von $G = (V_G, E_G)$ wie im obigen Lemma definiert mit $V_H = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ und $V_G = \{u_1, \ldots, u_n\}$. Berechne ein kleinstes Vertex Cover $T = \{x_{i_1}, \ldots, x_{i_k}, y_{j_1}, \ldots, y_{j_l}\}$ für H mit einem Matching Algorithmus für bipartite Graphen.
Sei $T' = \{u_{i_1}, \ldots, u_{i_k}, u_{j_1}, \ldots, u_{j_l}\}$. Wir zeigen

1. $|T'| = |T|$
   Angenommen, es gilt $i_r = j_s = b$ für zwei Indizes i_r und j_s. Da T ein kleinstes Vertex Cover für H ist, folgt:

   x_b ist mit mindestens einem Knoten $y_c \in \{y_1, \ldots, y_n\} \setminus T$ adjazent. y_b ist mit mindestens einem Knoten $x_a \in \{x_1, \ldots, x_n\} \setminus T$ adjazent.

   Also gibt es in der transitiven Orientierung von G die Kanten (u_a, u_b) , (u_b, u_c) und somit auch $u_a, u_c)$, welche von T nicht überdeckt wird. Widerspruch.

2. $S = V_G \setminus T'$ ist Independent Set für G
   Gäbe es eine Kante $\{u_a, u_b\}$ in S (sei (u_a, u_b) die entsprechende Kante in der transitiven Orientierung für G), so würde T die Kante $\{x_a, y_b\}$ in H nicht überdecken. Widerspruch.

3. S ist ein Maximum Independent Set für G
   Es gilt: $|S| = n - |T'| = n - |T| = n - \tau(H)$. Außerdem gilt $\tau(H) = \nu(H) = n - \kappa(G)$. Somit folgt $|S| = \kappa(G) \geq \alpha(G)$

□

Definition. Ein Graph $G = (V, E)$ ist *stark perfekt*, falls jeder induzierte Teilgraph H von G ein Independent Set $S \subseteq V_H$ besitzt, das aus jeder maximalen Clique in H mindestens einen Knoten enthält.

Bemerkung. Stark perfekte Graphen sind perfekt. (Übungsaufgabe)

Perfekte Graphen sind insbesondere aus algorithmischer Sicht sehr interessant. Für perfekte Graphen sind alle Probleme, die mit den Graphparametern $\chi(G), \omega(G), \kappa(G)$ und $\alpha(G)$ verknüpft sind, in polynomieller Zeit lösbar. [9]
Das Erkennungsproblem für perfekte Graphen ist offen.

[9] z.B. mit der Ellipsoidmethode aus der linearen Optimierung

15 Separatoren und planare Graphen

Da jeder planare Graph einen Knoten vom Grad höchstens fünf besitzt, hat jeder planare Graph (mit mindestens sieben Knoten) einen Knotenseparator mit höchstens fünf Knoten.

Definition. Der *Durchmesser* eines Graphen G ist die Länge des längsten kürzesten Weges in G,

$$d(G) = \max\{\text{Länge}(P) \mid P \text{ ist kürzester Weg in } G\}.$$

Lemma 15.1. *Sei $G = (V, E)$ ein planarer Graph und $B = (V, E')$ ein Spannbaum für G mit Durchmesser s. Dann besitzt G einen balancierten Knotenseparator der Größe s.*

Beweis. Betrachte eine planare Einbettung für G und sei o.B.d.A. G trianguliert. Sei $e \in E \setminus E'$ eine Kante in G, welche nicht im Spannbaum ist. Die Kante e bildet mit einigen Kanten aus E' einen eindeutig bestimmten Kreis $C(e)$ in G. Sei $\text{int}(e)$ und $\text{ext}(e)$ die Menge der Knoten im innerhalb bzw. außerhalb des Kreises in der planaren Einbettung für G.
Wir suchen eine Kante e, so daß $\text{int}(e)$ und $\text{ext}(e)$ höchstens $\frac{2}{3}|V_G|$ Knoten enthalten.
Sei $|\text{int}(e)| > |\text{ext}(e)|$. Solange $|\text{int}(e)| > \frac{2}{3}|V_G|$, suchen wir eine neue Kante e' mit $e \cap e' \neq \emptyset$ und

1. $\text{int}(e') \subseteq \text{int}(e)$ und im Inneren von $C(e')$ ist mindestens ein Face weniger als im Inneren von $C(e)$ und
2. $\text{ext}(e') \leq \frac{2}{3}|V_G|$.

Bei der Ersetzung von e durch e' muß $\text{int}(e')$ nicht echt kleiner als $\text{int}(e)$ sein. Da die planare Einbettung $O(n)$ viele Faces hat, terminiert der Algorithmus nach $= (n)$ vielen Schritten. Sei $e = \{u, v\}$. Betrachte das eindeutige Face mit den Knoten u, v, w im Kreis $C(e)$. Wegen $\text{int}(e) > 0$ können die Kanten $\{u, w\}$ und $\{v, w\}$ auf dem Kreis $C(e)$ liegen. Wir unterscheiden folgende Fälle:

1. Genau eine der beiden Kanten liegt auf dem Kreis. Sei o.B.d.A. die Kante $e = \{v, w\}$ auf dem Kreis $C(e)$. Dann ist $e' = \{u, w\}$ keine Baumkante und $\text{int}(\{u, w\}) \subseteq \text{int}(\{u, v\})$.
2. Keine der beiden Kanten liegt auf dem Kreis $C(e)$.
   (a) Genau eine der beiden Kanten ist eine Baumkante (o.B.d.A. sei $\{u, w\}$ keine Baumkante). Dann enthält $\text{int}(\{u, v\})$ alle Knoten aus $\text{int}(\{v, w\})$. Damit ist $\text{int}(\{v, w\}) \subseteq \text{int}(\{u, v\})$.

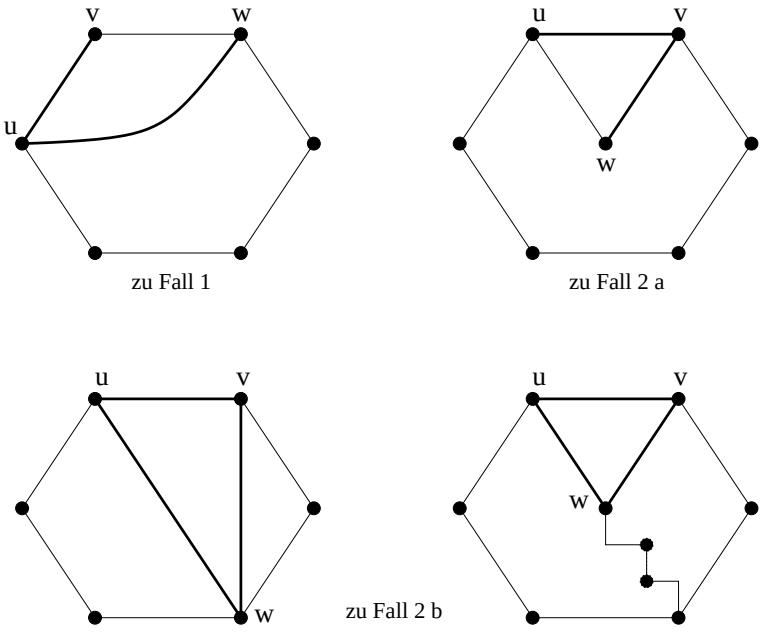


Abbildung 15.1: Zur Fallunterscheidung im Beweis des Lemmas. Kanten, die nicht im Spannbaum liegen, sind fett gezeichnet.

(b) Keine der beiden Kanten ist eine Baumkante. Sei $\left|\text{int}(\{u,w\})\right| \geq \left|\text{int}(\{v,w\})\right|$. Dann wähle $e' = \{u,w\}$. Dabei nimmt die Anzahl der Faces im Inneren mindestens um zwei ab. Ferner gilt:

$$\begin{aligned}
\frac{2}{3}n &< \left|\text{int}(\{u,v\})\right| \\
&= \left|\text{int}(\{v,w\})\right| + \left|\text{int}(\{v,w\})\right| + \left|C(\{u,w\}) \cap C(\{v,w\})\right| - 1 \\
&\leq 2 \cdot \left|\text{int}(\{u,w\})\right| + \left|C(\{u,w\})\right| \\
&\leq 2 \cdot \left|\text{int}(\{u,w\})\right| + 2 \cdot \left|C(\{u,w\})\right|
\end{aligned}$$

Daraus folgt:

$$\begin{aligned}
\left|\text{ext}(\{u,w\})\right| &= n - \left|C(\{u,w\})\right| - \left|\text{int}(\{u,w\})\right| \\
&< n - \frac{1}{3}n = \frac{2}{3}n
\end{aligned}$$

□

Satz 15.1. *Sei G ein planarer Graph mit n Knoten. Dann besitzt G einen balancierten Knotenseparator der Größe $2\sqrt{2n}$.*

Beweis. Wähle einen Knoten w und teile alle übrigen Knoten in Mengen $S_1, \ldots, S_l$ auf, so daß in der Menge S_i alle Knoten mit Abstand i zum Knoten w liegen. Die Menge S_0 enthält nur den Knoten w.
Falls $2l \leq 2\sqrt{2n}$, dann gibt es einen Spannbaum mit Durchmesser $\leq 2\sqrt{2n}$ und aufgrund des obigen Lemmas folgt die Aussage.

Falls $2l > 2\sqrt{2n}$, dann gruppieren wir die Mengen S_i nach ihrem Rest modulo einer später zu bestimmenden Zahl $s < l$. Sei also

$$L_j = \bigcup_{\substack{i \equiv j \mod s \\ 0 \leq i \leq l}} S_i \text{ für } 0 \leq j \leq s$$

Dann gibt es mindestens ein k mit $0 \leq k < s$, so daß L_k höchstens $\lfloor \frac{n}{s} \rfloor$ Knoten enthält.

Fall 1: Wenn jede Zusammenhangskomponente in $G \setminus L_k$ höchstens $\frac{2}{3}$ Knoten hat, dann ist L_k ein balancierter Knotenseparator.

Fall 2: Angenommen, es gibt eine Zusammenhangskomponente H in $G \setminus L_k$ mit mehr als $\frac{2}{3}n$ Knoten. Die Knoten von H sind in höchstens $s-1$ aufeinanderfolgenden Mengen S_i enthalten. Wir zeigen nun, daß H in einen planaren Graphen mit Durchmesser $2s$ eingebettet werden kann. Sei $i_{\min}$ der kleinste Index, so daß $S_{i_{\min}} \cap V_H \neq \emptyset$. Falls $i_{\min} = 0$, dann enthält H bereits den Knoten w und wir sind fertig. Ist $i_{\min} > 0$, dann verbinden wir die Knoten aus $S_{i_{\min}} \cap H$ mit einem neuen Knoten w', d.h. wir kontrahieren alle Knoten aus den Mengen $S_1 \cup S_2 \cup \ldots \cup S_{i_{\min}-1}$ zu einem Knoten w'. Der dabei entstehende Graph ist planar und hat einen Durchmesser von höchstens $2s$. Aus dem vorherigen Lemma folgt, daß es für H einen balancierten Separator C' mit höchstens $2s$ Knoten gibt.

Wenn wir $C = C' \cup L_k$ wählen, dann ist C ein balancierter Knotenseparator für G der Größe $|C| \leq \lfloor \frac{n}{s} \rfloor + 2s$. Die Summe wird für $s = \lceil \sqrt{\frac{n}{2}} \rceil$ minimal. Damit gilt $|C| \leq \sqrt{2n} + \sqrt{2n} = 2\sqrt{2n}$. □

Die Größe des Separators im vorherigen Satz ist bis auf einen multiplikativen Faktor auch eine untere Schranke.

Satz 15.2. *Sei $G_{k,k} = (V, E)$ der $k \times k$ Gittergraph mit der Knotenmenge $V = \{1, \ldots, k\} \times \{1, \ldots, k\}$ und der Kantenmenge*

$$E = \left\{ \{(x_1, y_1), (x_2, y_2)\} \middle| x_1 = x_2, |y_1 - y_2| = 1 \text{ oder } y_1 = y_2, |x_1 - x_2| = 1 \right\}$$

Wir sagen Knoten (x, y) ist in Zeile x und in Spalte y. Sei $n = k^2$. Dann besitzt G keinen balancierten Knotenseparator C, so daß

1. *die Komponenten in $G \setminus C$ höchstens αn Knoten haben für ein $0 < \alpha < 1$ und*
2. *C höchstens $\beta\sqrt{n}$ Knoten hat für ein $\beta < \min\left\{\frac{1}{2}, \sqrt{1-\alpha} - \frac{1}{2\sqrt{n}}\right\}$*

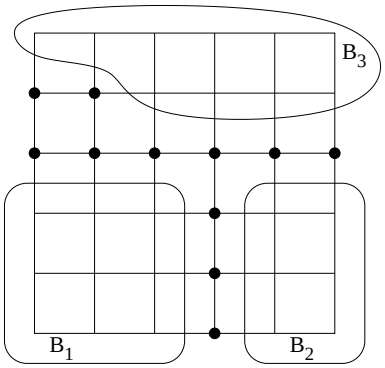


Abbildung 15.2: Ein 6×6 Gitter mit einem Separator, der den Graphen in 3 Komponenten teilt

Beweis. Sei $C \subseteq V$ ein Knotenseparator für $G_{k,k}$, so daß alle Komponenten in $G_{k,k} \setminus C$ höchstens αn Knoten haben für ein $0 < \alpha < 1$. Wir zeigen, daß dann $|C| > \beta\sqrt{n}$.
Zunächst bestimmen wir eine Menge $A \subseteq V \setminus C$ der Größe $\beta^2 n < |A| < \frac{n}{2}$ mit der Eigenschaft, daß alle Nachbarn der Knoten in $C \cup A$ sind.

1. Hat $G \setminus C$ eine Komponente B_i mit mindestens $\frac{n}{2}$ Knoten, dann definiere $A = V \setminus (B_I \cup C)$. Nun gilt $\frac{n}{2} \geq |A| \geq n - \alpha n - \beta\sqrt{n} > \beta^2 n$.

2. Sind alle Komponenten B_i in $G \setminus C$ kleiner als $\frac{n}{2}$, dann definiere a als Vereinigung von Komponenten von $G \setminus C$, so daß $\frac{n}{4} < |A| < \frac{n}{2}$

Sei z die Anzahl der Zeilen und s die Anzahl der Spalten von $G_{k,k}$, die mindestens einen Knoten aus A haben. Sei o.B.d.A. $s \leq z$.
Sei z^* die Anzahl der Zeilen, die nur Knoten aus A enthalten. Offenbar muß C mindestens $z - z^*$ Knoten haben.
Falls $z* > 0$, dann ist $s = k$. Wegen $s \leq z \leq k$ ist dann auch $z = k$. Wegen $kz^* \leq |A| \leq \frac{n}{2}$ folgt $z^* \leq \frac{k}{2}$ und somit $|C| \geq z - z^* \geq k - \frac{k}{2} > \beta\sqrt{n}$.
Falss $z^* = 0$, dann folgt wegen $\beta^2 n < |A| \leq sz \leq z^2$ ebenfalls $|C| \geq z > \beta\sqrt{n}$. □

Aus diesem Satz folgt zum Beispiel, daß es für planare Graphen im allgemeinen keinen balancierten Knotenseparator der Größe $(\frac{1}{2} - \varepsilon)\sqrt{n}$ gibt, so daß jede Komponente höchstens $\frac{n}{3}$ Knoten hat.

Maximum Independent Set für planare Graphen

Satz 15.3. *Das Maximum Independent Set Problem für planare Graphen ist NP-vollständig.*

Beweis. Reduktion von Maximum Independent Set für allgemeine Graphen. Sei $G = (V, E)$ und ein $s \in \mathbb{N}$ gegeben. Betrachte eine Einbettung von G in der Ebene, so daß [10]

[10] vgl. mit dem Beweis zur NP-Vollständigkeit der 3-Färbbarkeit von planaren Graphen

1. Kanten sich nicht selbst schneiden,

2. jede Kante eine andere Kante höchstens einmal schneidet und

3. sich in jedem Punkt höchstens zwei Kanten schneiden.

Ersetze eine Kreuzung von zwei Kanten $\{u, v\}$ und $\{x, y\}$ durch den Graphen J in Abbildung 15.3. Die übrigen Kreuzunge der Kanten $\{u, v\}$ bzw. $\{x, y\}$ mit anderen Kanten (falls solche existieren) werden über die Kanten $\{u, u'\}$ $\{v, v'\}$ $\{x, x'\}$ $\{y, y'\}$ geführt. Sei G' das Ergebnis dieser Ersetzung. Wir zeigen unten, daß $\alpha(G') = \alpha 8G) + 6$. Wird die Ersetzungsoperation sukzessive für alle r Paare sich kreuzender Kanten durchgeführt, so erhalten wir einen planaren Graphen H mit $\alpha(H) = \alpha(G) + 6r$. es gilt also:

$$\alpha(G) \geq s \Leftrightarrow \alpha(H) - 6r \geq s.$$

Da die Transformation in polynomieller Zeit berechenbar ist, folgt die NP-Vollständigkeit. *Beweis der Gleichung* $\alpha(G') = \alpha(G) + 6$

$\geq$: Sei S eine unabhängige Menge für G. Dann enthält S höchstens je einen Knoten der Kanten $\{u, v\}$, $\{x, y\}$.

Falls S keinen der beiden Knoten v, y enthält, dann ist $S \cup \{a, l, v', y', g, i\}$ eine unabhängige Menge für G'.

Falls S keinen der beiden Knoten v, x enthält, dann ist $S \cup \{b, i, v', x', f, l\}$ eine unabhängige Menge für G'.

Die beiden anderen Fälle, daß S keinen der Knoten u, y bzw. keinen der Knoten u, x enthält sind völlig symmetrisch zu den beiden ersten Fällen.

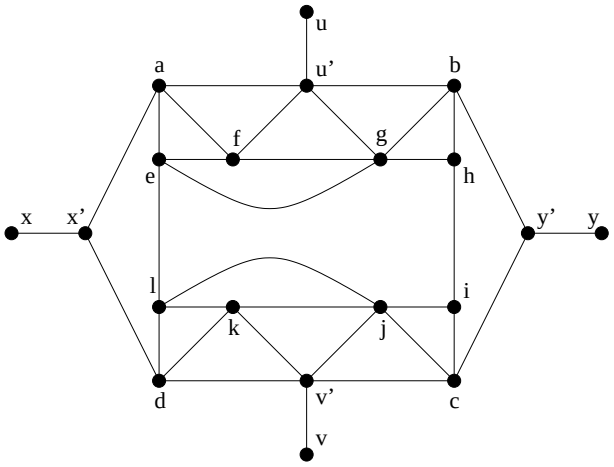


Abbildung 15.3: Graph J im Beweis zur NP-Vollständigkeit des Maximum Independent Set Problems

$\leq$: Sei T eine unabhängige Menge für G'. Wir zeigen, daß durch Streichung von höchstens sechs Knoten aus $T \cap W$ mit W={a,b,. . . ,l,u,u',v,v',x,x',y,y'} eine unabhängige Menge für G entsteht.

Fall 1: $|T \cap \{u, v\}| \leq 1$ und $|T \cap \{x, y\}| \leq 1$. Dann kann T aus den Kreisen $\{x', a, e, l, d\}$ und $\{y', b, h, i, c\}$ jeweils höchstens zwei Knoten und aus den Kreisen $\{u', f, g\}$ und $\{v', k, j\}$ höchstens einen Knoten enthalten. Da $T \setminus W'$ mit $W' = W \setminus \{u, v, x, y\}$ eine unabhängige Menge für G ist, genügt es, diese sechs Knoten aus T zu entfernen.

Fall 2: $u, v, x, y \in T$. Da $T \cap \{u', v', x', y'\} = \emptyset$ und T aus jedem der vier Dreiecke $\{a, e, f\}$, $\{b, g, h\}$, $\{l, k, d\}$ und $\{i, j, c\}$ höchstens einen Knoten enthalten kann, folgt $|T \cap W'| \leq 4$. Entferne die vier Knoten aus den Dreiecken und die beiden Knoten x und u.

Fall 3: $|T \cap \{u, v\}| \leq 1$ und $|T \cap \{x, y\}| = 2$. Dann können x' und y' nicht in T sein. Wieder enthält T aus den vier Dreiecken höchstens je einen Knoten. Sind beide Knoten u', v' in T, so kann in T höchstens je ein Knoten aus den Kanten $\{e, l\}$ und $\{h, i\}$ in T sein. Also gilt $|T \cap W'| \leq 4$. Ist nur einer der beiden Knoten u', v' in T, dann ist $|T \cap W'| \leq 5$. Entferne diese vier bzw. fünf Knotenund Knoten x aus T.

Fall 4: $|T \cap \{u, v\}| = 2$ und $|T \cap \{x, y\}| \leq 1$. Dann können u' und v' nicht in T sein. Wieder enthält T aus den vier Dreiecken höchstens je einen Knoten. Ist nur einer der beiden Knoten x', y' in T, so ist $|T \cap W'| \leq 5$. Sind beide Knoten x', y' in T, so kann T höchstens noch einen Knoten aus den Dreiecken $\{e, f, g\}$ und $\{l, k, j\}$ sowie aus der Kante $\{h, i\}$ haben. Also gilt: $|T \cap W'| \leq 5$. Entferne diese Knoten und Knoten u. □

Eine maximale unabhängige Menge (Independent Set) von $G = (V, E)$ mit $|V| = n$ kann in schwach exponentieller Zeit $2^{O(\sqrt{n})}$ berechnet werden.
Berechne einen Knotenseparator C für G mit höchstens $2\sqrt{2n}$ Knoten, so daß jede Komponente in $G \setminus C$ höchstens $\frac{2}{3}n$ Knoten hat. Für jede unabhängige Knotenmenge S in $G(C)$ wird in jeder Komponente von $G \setminus C$ rekursiv eine größte unabhängige Menge berechnet, die keinen Nachbarn in S hat. Diese unabhängigen Mengen und S werden zu einer in G unabhängigen Menge I zusammengefasst. Die größte Menge I ist eine größte unabhängige Menge für G.
Laufzeitabschätzung: Es gibt $\leq 2^{\sqrt{8n}}$ viele Teilmengen von C. Also gilt

$$T(n) \leq f(n) + 2^{\sqrt{8n}} \cdot g(n) \cdot T\Big(\frac{2}{3}n\Big).$$

Dabei ist $f(n)$ der Zeitaufwand für das Finden eines Knotenseparators mit höchstens $\sqrt{8n}$ Knoten und $g(n)$ der Zeitaufwand für das Überprüfen, ob S unabhängig ist und das Entfernen der Nachbarn aus S. Also ist $T(n) \in 2^{O(\sqrt{n})}$

$T(n) \leq O(n) + 2^{\sqrt{8n}} \cdot O(n) \cdot T\left(\frac{2}{3}n\right) \in 2^{O(\sqrt{n})}$, denn:
Seien $K_1, K_2 \in \mathbb{R}$, so daß die Funktionen f und g durch $K_1 n$ bzw. $K_2 n$ beschränkt sind. Sei ferner n_0 derart, daß $(K_1 + K_2)n \leq 2^{\sqrt{8n}}$ für alle $n \geq n_0$ und L eine Konstante, so daß $T(n) \leq L$ für alle $n < n_0$. Wir zeigen durch Induktion, daß

$$T(n) \leq L \cdot 2^{\left(\frac{2\sqrt{8n}}{1-\sqrt{2/3}}\right)}.$$

Für alle $n < n_0$ stimmt die Aussage.
Für $n \geq n_0$ gilt

$$\begin{aligned} T(n) &\leq K_1 \cdot n + 2^{\sqrt{8n}} \cdot K_2 \cdot n \cdot T\left(\frac{2}{3}n\right) \\ &\leq 2^{2\sqrt{8n}} \cdot T\left(\frac{2}{3}n\right) \\ &\leq 2^{2\sqrt{8n}} \cdot L \cdot 2^{\left(\frac{2\sqrt{2/3 \cdot 8n}}{1-\sqrt{2/3}}\right)} \\ &= L \cdot 2^{\left(\frac{2\sqrt{8n}}{1-\sqrt{2/3}}\right)} \end{aligned}$$

16 Graphen mit beschränkter Cliquenweite

16.1 Definitionen und Einführung

Wir betrachten knotenmarkierte Graphen $G = (V_G, E_G, \text{lab}_G)$ mit $\text{lab} : V_G \to \mathbb{N}$.
Bezeichnungen. Für eine Zahl $k \in \mathbb{N}$ sei $[k] := \{1, \ldots, k\}$. Den markierten Graphen, der aus genau einem Knoten besteht, der mit t markiert ist, bezeichnen wir mit $\bullet_t$.

Definition. Sei $k \in \mathbb{N}$. Die Klasse CW_k ist wie folgt definiert:

1. Der Graph $\bullet_t$ für $t \in [k]$ ist in CW_k.
2. Seien $G = (V_G, E_G, \text{lab}_G) \in CW_k$ und $J = (V_J, E_J, \text{lab}_J) \in CW_k$ zwei Knotendisjunkte markierte Graphen. Dann ist $G \oplus J := (V', E', \text{lab}')$ definiert durch $V' = V_G \cup V_J$ und $E' = E_G \cup E_J$ und
$$\text{lab}'(u) = \begin{cases} \text{lab}_G(u) & \text{falls } u \in V_G \\ \text{lab}_J(u) & \text{falls } u \in V_J \end{cases} \quad \forall u \in V'$$
ebenfalls in CW_k.
3. Seien $i.j \in [k]$ zwei verschiedene Zahlen und $G = (V_G, E_G, \text{lab}_G) \in CW_k$, dann sind
   (a) $\varrho_{i \to j}(G) = (V_G, E_G, \text{lab}')$ definiert durch
   $$\text{lab}'(u) = \begin{cases} \text{lab}_G(u) & \text{falls } \text{lab}_G(u) \neq i \\ j & \text{falls } \text{lab}_G(u) = i \end{cases} \quad \forall u \in V_G$$
   und
   (b) $\eta_{i,j}(G) = (V_G, E', \text{lab}_G)$ definiert durch
   $$E' = E_G \cup \big\{ \{u, v\} \;\big|\; u, v \in V_G, \text{lab}_G(u) = i, \text{lab}_G(v) = j \big\}$$
   ebenfalls in CW_k.

Die *Cliquenweite* von G ist die kleinste Zahl k, so daß $G \in CW_k$.

Beispiel. $CW_1 = \big\{ \bullet_1 \quad , \quad \bullet_1 \bullet_1 \quad , \quad \bullet_1 \bullet_1 \bullet_1 \, , \quad \ldots \big\}$
In CW_2 sind unter anderem alle vollständigen Graphen und alle vollständig bipartiten Graphen enthalten (vgl. Abbildung 16.1).

Ein Ausdruck, der mit den Operationen $\bullet_t, \oplus, \varrho_{i \to j}$ und $\eta_{i,j}$ mit $t, i, j \in [k]$ gebildet wurde, nennen wir einen k-Ausdruck.

Definition. NLC-Weite (node label controlled)

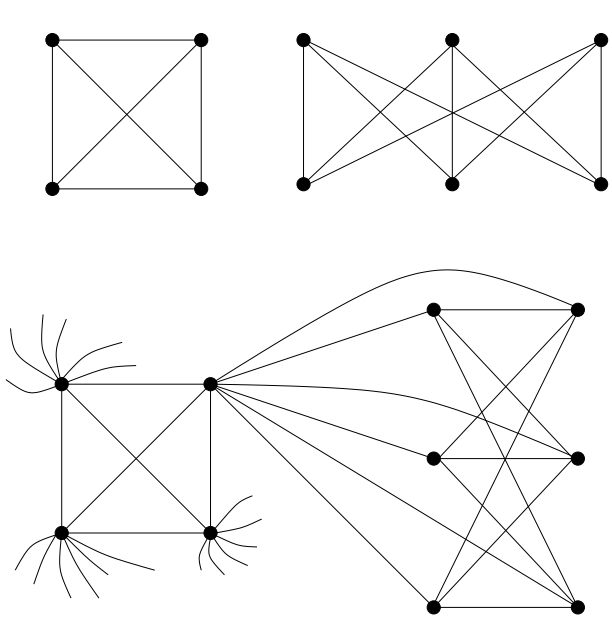


Abbildung 16.1: Beispiele für Graphen in CW_2

1. $\bullet_t$ für $t \in [k]$ ist in NLC_k.

2. Seien $G = (V_G, E_G, \mathrm{lab}_G) \in NLC_k$ und $J = (V_J, E_J, \mathrm{lab}_J) \in NLC$ zwei knotendisjunkte markierte Graphen und $S \subseteq [k] \times [k]$. Dann ist $G \times_S J = (V', E', \mathrm{lab}')$ definiert durch $V' = V_G \cup V_J$,

$$E' = E_G \cup E_J \cup \left\{ \{u, v\} \;\middle|\; u \in V_G, v \in V_J, \big(\mathrm{lab}_G(u), \mathrm{lab}_J(v)\big) \in S \right\} \text{ und}$$

$$\mathrm{lab}'(u) = \begin{cases} \mathrm{lab}_G(u) & \text{falls } u \in V_G \\ \mathrm{lab}_J(u) & \text{falls } u \in V_J \end{cases} \forall u \in V'$$

   ebenfalls in NLC_k.

3. Sei $G = (V_G, E_G, \mathrm{lab}_G) \in NLC_k$ und $R : [k] \to [k]$, dann ist $\circ_R(G) = (V_G, E_G, \mathrm{lab}')$ definiert durch

$$\mathrm{lab}'(u) = R(\mathrm{lab}_G(u)) \; \forall u \in V_G$$

   ebenfalls in NLC_k.

Die *NLC-Weite* von G ist die kleinste Zahl k, so daß $G \in NLC_k$.

Ein Ausdruck, der mit den Operationen $\bullet_t, \times_S, \circ_R$ mit $S \subseteq [k]^2$ und $R : [k] \to [k]$ gebildet wurde, nennen wir einen k-Ausdruck.
Ein unmarkierter Graph $G = (V, E)$ hat Cliquenweite bzw. NLC-Weite k, wenn es eine Markierungsfunktion $\mathrm{lab} : V \to [k]$ gibt, so daß (V, E, lab) Cliquenweite bzw. NLC-Weite k hat.

Lemma 16.1. *Die Menge der Graphen in NLC_1 ist genau die Menge der Cographen.*

Beweis. Finde zu den erlaubten Operationen bei der Bildung von Cographen die entsprechenden Operationen für NLC_1-Graphen und umgekehrt.

Cographen	NLC-Weite 1	NLC-Weite 1	Cographen
$\bullet$	$\bullet_1$	$\bullet_1$	$\bullet$
$G \cup J$	$G \times_\emptyset J$	$\circ_R G$	G
$\overline{\bullet}$	$\overline{\bullet}$	$G \times_\emptyset J$	$G \cup J$
$\overline{\overline{G'}}$	G'	$G \times_{\{(1,1)\}} J$	$\overline{\overline{G} \cup \overline{J}}$
$\overline{G' \cup J'}$	$\overline{G'} \times_{\{(1,1)\}} \overline{J'}$		

Es gilt für alle $R : [1] \to [1]$: $\circ_R(G) = G$. □

Bemerkung. Es gilt:

1. $CW_2 = NLC_1 = \{\text{Cographen}\}$
2. $CW_k \subsetneq CW_{k+1}$, $NLC_k \subsetneq NLC_{k+1}$
3. $CW_k \subseteq NLC_k \subseteq CW_{2k}$

Beispiel. (vgl. Abbildung 16.2)

$$\begin{aligned} G_1 &= \bullet_1 \times_{\{(1,2)\}} \bullet_2 \\ G_2 &= G_1 \times_{\{(1,1)\}} \bullet_1 \\ G_3 &= G_2 \times_\emptyset G_2 \\ G_4 &= G_3 \times_\emptyset G_3 \\ G_5 &= G_4 \times_{\{(2,2)\}} \bullet_2 \end{aligned}$$

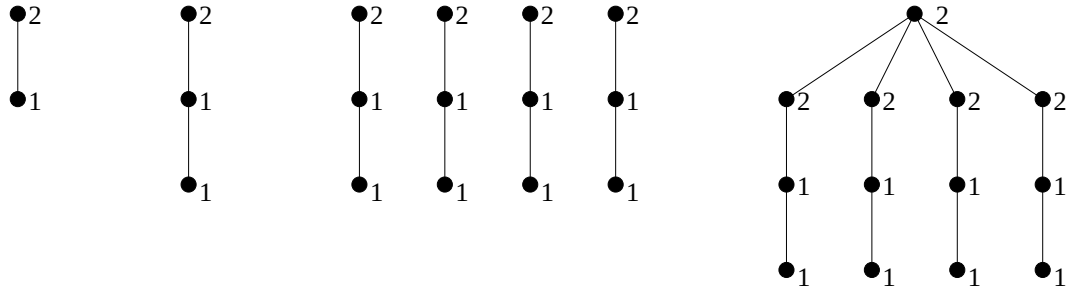


Abbildung 16.2: Aufbau des Graphen G_5 im obigen Beispiel

$$
\begin{aligned}
G_1 &= \bullet_1 \times_{\{(1,2)\}} \bullet_2 \\
G_2 &= \circ_R(G_1 \times_{\{(2,3)\}} \bullet_3) \text{ mit } R(1)=1,\ R(2)=1,\ R(3)=2 \\
&\dots \\
G_i &= \circ_R(G_{i-1} \times_{\{(2,3)\}} \bullet_3) \text{ mit } R(1)=1,\ R(2)=1,\ R(3)=2
\end{aligned}
$$

Lemma 16.2. *Die Klasse NLC_k ist abgeschlossen bzgl. Kantenkomplementierung.*

Beweis. Für alle $t \in [k]$, $R : [k] \to [k]$, $S \subseteq [k] \times [k]$ und $G, J \in NLC_k$ gilt:

$$\overline{\bullet_t} = \bullet_t$$

$$\overline{\circ_R(G)} = \circ_R(\overline{G})$$

$$\overline{G \times_S J} = \overline{G} \times_{\overline{S}} \overline{J}$$

mit $\overline{S} = \big\{(i,j) \in [k]^2 \;\big|\; (i,j) \notin S\big\}$. □

Die Klassen CW_k und NLC_k sind abgeschlossen bzgl. induzierten Teilgraphen, aber nicht bzgl. aller Teilgraphen.
Der Ausdrucksbaum $T = (V_T, E_T, \mathrm{lab}_T)$ für einen k-Ausdruck X ist ein geordneter Baum mit einer Wurzel, dessen Knoten mit den Operationen des Ausdrucks markiert sind, und dessen Kanten von den Söhnen zu den Vätern gerichtet sind.

$$\mathrm{lab}_T : V_T \to \big\{\bullet_t \;\big|\; t \in [k]\big\} \cup \big\{\eta_{i,j} \;\big|\; i,j \in [k]\big\} \cup \big\{\varrho_{i\to j} \;\big|\; i,j \in [k]\big\} \cup \{\oplus\} \text{ bzw.}$$

$$\mathrm{lab}_T : V_T \to \big\{\bullet_t \;\big|\; t \in [k]\big\} \cup \big\{X_S \;\big|\; S \subseteq [k]^2\big\} \cup \big\{\circ_R \;\big|\; R : [k] \to [k]\big\}$$

Der Ausdrucksbaum T für $\bullet_t$ ist ein einzelner Knoten r (die Wurzel von T), der mit $\bullet_t$ markiert ist.
Der Ausdrucksbaum T für $\eta_{i,j}(X)$ bzw. $\varrho_{i\to j}(X)$ besteht aus dem Baum T' für den Ausdruck X und einem zusätzlichen Knoten r (die Wurzel von T) markiert mit $\eta_{i,j}$ bzw. $\varrho_{i\to j}$ mit einer Kante von der Wurzel von T' zu r.
Der Ausdrucksbaum T für $X_1 \oplus X_2$ besteht aus der disjunkten Vereinigung der Bäume T_1 und T_2 für X_1 und X_2 und einem zusätzlichen Knoten r (die Wurzel von T) markiert mit $\oplus$ und zwei zusätzlichen Kanten von der Wurzel von T_1 und der Wurzel von T_2 zu Knoten r. Die Wurzel von T_1 bzw. von T_2 ist der linke bzw. rechte Sohn von r.
Analog für $NLC-k$-Ausdrücke.

Satz 16.1. $CW_k \subseteq NLC_k$

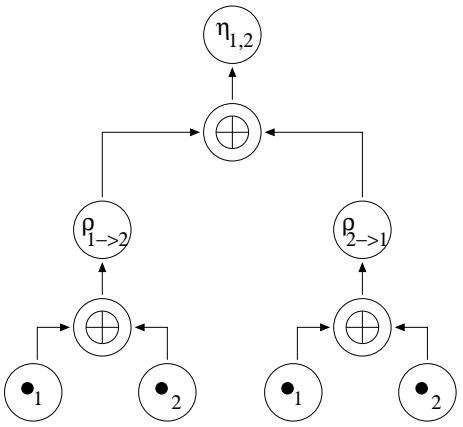


Abbildung 16.3: Der Ausdruckbaum für den $CW-2$-Ausdruck

Beweis. Sei $G \in CW_k$ und X ein Ausdruck, der G definiert. X kann einfach verändert werden, so daß jede kante $\{u, v\}$ in G direkt nach der disjunkten Vereinigung der beiden Graphen G_1 und G_2 erzeugt wird, die u bzw. v enthalten. Jeder Ausdruck kann so umgeformt werden, daß nach einer $\oplus$-Operation erst einmal ausschließlich $\eta_{i,j}$-Operationen und danach ausschließlich $\varrho_{i\to j}$-Operationen kommen. Das Ergebnis kann dann mit anderen Graphen disjunkt vereinigt werden.
Das bedeutet, daß alle $\eta_{i.j}$-Operationen nach einer $\oplus$-Operation mit einer $\times_S$-Operation realisiert werden können. Die nachfolgenden $\varrho_{i\to j}$-Operationen können alle mit einer $\circ_R$-Operation realisiert werden. □

Satz 16.2. $NLC_k \subseteq CW_{2k}$

Beweis. Induktiv über den Aufbau eines Graphen $G \in NLC_k$
Sei $G = \bullet_t$ $\surd$
Sei $G = \circ_R(G_1)$ $\surd$
Sei $G = G_1 \times_S G_2$. Markiere die Knoten $u \in G_2$ mit $\mathrm{lab}(u) + k$. Sei G_2' der ummarkierte Graph G_2. Bilde $G_1 \oplus G_2'$ und füge für alle $(i, j) \in S$ die Kanten zwischen Knoten ein, die mit i bzw. $k + j$ markiert sind. Anschließend markiere alle mit $i > k$ markierten Knoten mit $i - k$. $\surd$ □

Satz 16.3. *Ein unmarkierter Graph $G = (V, E, lab)$ mit $lab(u) = 1$ für alle $u \in V$ mit einer Baumweite von k hat eine NLC-Weite von $2^{k+1} - 1$.*

Beweis. Sei $G = (V, E, \mathrm{lab})$ ein Graph mit Baumweite k und $H = (V, E_H, \mathrm{lab})$ mit $E \subseteq E_H$ ein kantenmaximaler Graph mit Baumweite k. D.h.: Für jede Kantenmenge E' mit $E_H \subsetneq E'$ hat (V, E') eine Baumweiter $> k$.
Sei $o = (v_1, \ldots, v_n)$ eine perfekte Eliminationsordnung für H.[11] Definiere folgende

[11] Diese existiert, da H chordal ist.

Mengen:

$$N^+(G,o,i) = \big\{ v_j \;\big|\; \{v_i,v_j\} \in E, i<j \big\}$$
$$N^-(G,o,i) = \big\{ v_j \;\big|\; \{v_i,v_j\} \in E, i>j \big\}$$

Die Knoten in $N^+(G,o,1),\ldots,N^+(G,o,n)$ induzieren Cliquen in H mit höchstens k Knoten. Die Struktur von H kann durch den Baum $T(H,o)=(V,E_T)$ mit

$$E_T = \big\{ \{v_i,v_j\} \in E_H \;\big|\; i<j \text{ und falls } i<j'<j \text{ für ein } j', \text{ dann gilt } \{v_i,v_{j'}\} \notin E_H \big\}$$

dargestellt werden.
$T(H,o)$ ist ein Baum, weil $T(H,o)$ zusammenhängend ist und es für jeden Knoten v_i aus $T(H,o)$ höchstens eine Kante $\{v_i,v_j\} \in E_H$ mit $i<j$ gibt.
Sei $\mathrm{col} : V_H \to [k+1]$ eine $k+1$-Färbung für H. Wir definieren C_i als die Menge der Farben der Knoten in $N^+(G,o,i)$. Sei b eine Bijektion $b : 2^{[k+1]} \to [2^{k+1}]$, $b(C_i) \in [2^{k+1}]$.
Wir definieren einen Graphen J_i für $i=1,\ldots,n$ mit NLC-Weite $2^{k+1}-1$.

1. Falls $N^-\big(T(H,o),o,i\big) = \emptyset$, dann sei $J_i = \bullet_{b(C_i)}$.
2. Falls $N^-\big(T(H,o),o,i\big) = \{v_{j_1},\ldots,v_{j_m}\}$, dann sei $J_i = o_R\big(\bullet_{b(C_i)} \times_S J\big)$, wobei

$$J = J_{j_1} \times_\emptyset \cdots \times_\emptyset J_{j_m}$$
$$S = \Big\{ \big(b(C_i), b(X)\big) \;\Big|\; X \subseteq [k+1] \wedge \mathrm{col}(v_i) \in X \Big\}$$
$$R\big(b(X)\big) = \big(X \setminus \mathrm{col}(v_i)\big) \text{ für alle } X \subseteq [k+1]$$

Der Graph J_n ist - abgesehen von der Markierung lab - der Graph G. Wir zeigen: Knoten w_l in J_i ist genau dann mit $b(X)$ markiert, wenn X die Menge aller Farben $\mathrm{col}(v_j)$ mit $\{v_l,v_j\} \in E_G$ und $l \le i < j$ ist. Induktion über i.

$i=1$: J_1 hat genau einen Knoten w_1, markiert mit $B(C_1)$, wobei $C_1 = \big\{\mathrm{col}(v_j) \;\big|\; \{v_1,v_j\} \in E_G, 1<j\big\}$

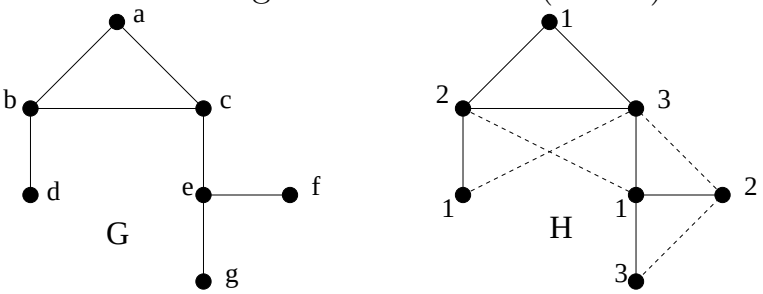
$i<1$: Wenn $N^-\big(T(H,o),o,v_i\big) = \emptyset$, dann hat J_i einen Knoten w_i, markiert mit $b(C_i)$, wobei $C_i = \big\{\mathrm{col}(v_j) \;\big|\; \{v_i,v_j\} \in E_G, i<j\big\}$

Wenn $N^-\big(T(H,o),o,v_i\big) = \{v_{j_1},\ldots,v_{j_m}\}$ mit $m>0$, dann ist J die disjunkte Vereinigung von $J_{j_1},\ldots,J_{j_m}$ und J_i ist definiert über die Vereinigung von $w_i = \bullet_{b(C_i)}$ und J. Die Komposition erzeugt genau dann eine Kante zwischen w_i und einem Knoten v_j aus J, markiert mit $b(X)$, wenn $\mathrm{col}(v_i) \in X$. (Aufgrund der Induktionsvoraussetzung folgt, daß J_i gleich G_i ist. [12]) Das Ummarkieren von J_i ändert jede Markierung $b(X)$ in $b\big(X \setminus \{\mathrm{col}(v_i)\}\big)$. Diese Ummarkierung ändert nicht die Markierung $b(C_i)$ von w_i, da $\mathrm{col}(v_i) \notin C_i$

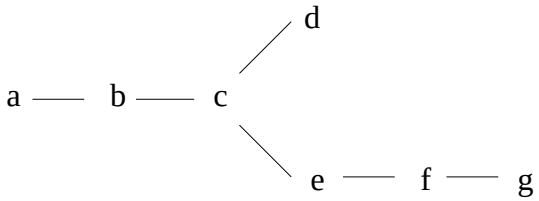
[12] bei Vernachlässigung der Markierungen; mit G_i ist der von den $v_1,\ldots,v_i$ induzierte Teilgraph von G gemeint.

□

Beispiel. Sei G der folgende Graph (links) und H ein zugehöriger kantenmaximaler Graph mit einer 3-Färbung der Knoten col (rechts).



Eine perfekte Eliminationsordnung für H ist $o = (g, f, e, d, c, b, a)$. Der zugehörige Baum $T(H, o)$ hat folgende Gestalt:



Wenn man $N^+(G, o, i)$ für $i = 1, \ldots, n$ betrachtet, erhält man folgende Mengen für C_i:

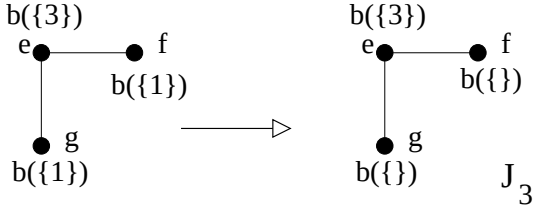
i	7	6	5	4	3	2	1
v_i	a	b	c	d	e	f	g
$N^+(G,o,i)$	$\{\}$	$\{a\}$	$\{a,b\}$	$\{b\}$	$\{c\}$	$\{e\}$	$\{e\}$
C_i	$\{\}$	$\{1\}$	$\{1,2\}$	$\{2\}$	$\{3\}$	$\{1\}$	$\{1\}$

Da $v_1 = g$ keinen Nachfolger in $T(H, o)$ hat, also $N^-\big(T(H, o), o, 1\big) = \emptyset$, ist $J_1 = \bullet_{b(\{1\})}$. Der Knoten ist mit $b(\{1\})$ markiert, da $N^+(G, o, 1) = \{e\}$ und $\mathrm{col}(e) = 1$.

Da $N^-\big(T(H, o), o, 2\big) = \{g = v_1\}$, entsteht J_2 durch disjunknte Vereinigung von J_1 mit $\bullet_{b(\{1\})}$. Kanten werden keine eingefügt, und eine Ummmarkierung findet auch nicht statt.

$J_3 = o_R\big(\bullet_{b(\{3\})} \times_S J_2\big)$. Mit $R(b(X)) = b(X \setminus \{1\})$ für alle $X \subseteq [k+1]$ und

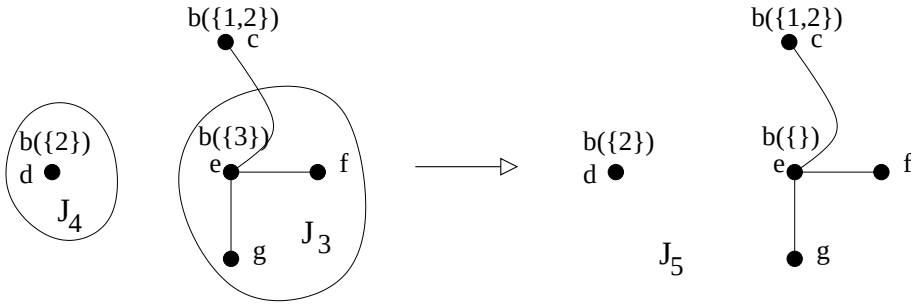
$$S = \Big\{\big(b(\{3\}), b(\{1\})\big),\ \big(b(\{3\}), b(\{1,2\})\big),\ \big(b(\{3\}), b(\{1,3\})\big),\ \big(b(\{3\}), b(\{1,2,3\})\big)\Big\}.$$



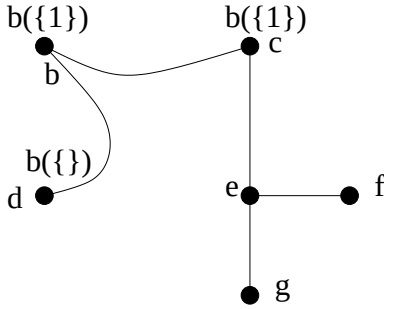
J_4 ist nur ein einzelner Knoten $\bullet_{b(\{2\})}$.

J_5 wird aus der disjunkten Vereinigung von J_3 und J_4 sowie einem weiteren Knoten gebildet: (Alte Markierungen $b(\{\})$, die für den weiteren Aufbau des Graphen

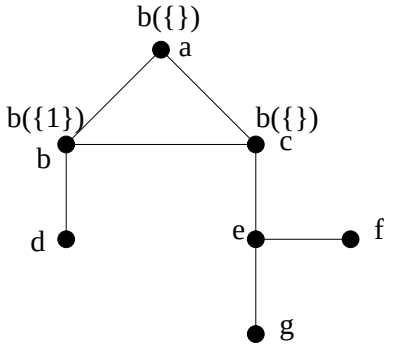
keine Rolle spielen, da zu diesen Knoten keine neuen Kanten führen können, werden der Übersicht wegen weggelassen.)



Fügt man einen weiteren Knoten ein, verbindet ihn mit den entsprechenden Knoten (bestimme dazu die Menge S) und markiert die Knoten entsprechend um (bestimme R), so erhält man J_6.



Und schließlich erhält man J_7, das bis auf die Markierungen der Knoten mit G identisch ist:



Definition. Für $n \geq 1$ sei $G_n = (V, E, \text{lab})$ der Graph mit

$$V = \{a_1, \ldots, a_n, b_0, \ldots, b_{2^n-1}\} \text{ und}$$

$$E = \big\{\{a_i, b_j\} \;\big|\; \text{in der Binärdarstellung von } j \text{ ist an der } i\text{-ten Stelle eine } 1\big\}$$

Satz 16.4. *Sei k die NLC-Weite von G_n, dann gilt $n-1 \leq k \leq n$.*
Sei k die Cliquenweite von G_n, dann gilt $n-1 \leq k \leq n+1$.

Beweis. Die oberen Schranken zeigt man, indem man zunächst die Knoten $a_1, \ldots a_{n-1}$ mit paarweise verschiedenen Labeln einfügt und dann *die* Hälfte der

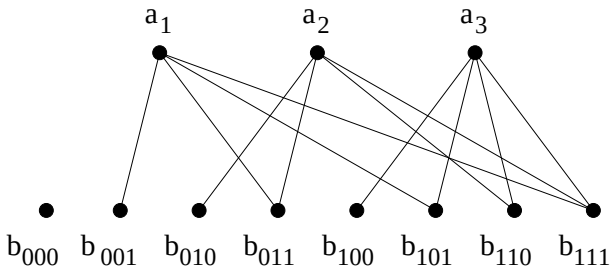


Abbildung 16.4: Der G_n für $n = 3$.

b-Knoten einfügt, die mit a_n verbunden sind. Diese sind mit dem letzten noch zur Verfügung stehenden Label markiert. Anschließend wird Knoten a_n eingefügt, mit den bereits eingefügten b-Knoten verbunden. Danach werden die übrigen b-Knoten eingefügt und entsprechend verbunden. Beispiel: G_3 hat NLC-Weite 3. Für Cliquenweite 3 kann man eine ähnliche Konstruktion angeben. [13]

$$G' = \Big(\Big(\big((a_1^1 \times_\emptyset a_2^2) \times_\emptyset b_{100}^3\big) \times_{\{(1,3)\}} b_{101}^3\Big) \times_{\{(2,3)\}} b_{110}^3\Big) \times_{\{(1,3),(2,3)\}} b_{111}^3$$

$$G_3 = \Big(\Big(\big((G' \times_{\{(3,3)\}} a_3^3) \times_\emptyset b_{000}^3\big) \times_{\{(1,3)\}} b_{001}^3\Big) \times_{\{(2,3)\}} b_{010}^3\Big) \times_{\{(1,3),(2,3)\}} b_{011}^3$$

Nachweis der unteren Schranken: Sei T ein Ausdrucksbaum für G_n mit $n > 1$. Für einen Knoten t aus T sei $G(t)$ der Teilgraph von G_n, der durch den Teilbaum $T(t)$ von T_G mit Wurzel t definiert wird. $T(t)$ enthält alle Knoten aus $T(G)$, für die es in T_G einen Weg zu t gibt.
Sei t so gewählt, daß $G(t)$ N_t b-Knoten enthält, mit $\frac{2^n}{4} \leq N_t < \frac{2^n}{2}$. Mehr als die Hälfte aller b-Knoten sind also nicht in $G(t)$, aber mehr als ein Viertel aller b-Knoten sind in $G(t)$ enthalten.
Alle a-Knoten in $G(t)$ müssen paarweise verschieden und auch von den b-Knoten in $G(t)$ verschieden markiert sein. Sei n_t die Anzahl der a-Knoten in $G(t)$. Die N_t b-Knoten in $G(t)$ müssen mindestens $\lceil \frac{N_t}{2^{n_t}} \rceil$ verschiedene Markierungen haben. Das Minimum für $n_t + \lceil \frac{N_t}{2^{n_t}} \rceil$ für $0 \leq n_t \leq n$ is $n - 1$. □

Bemerkung. Die Zeitkomplexität der Erkennung für NLC_k bzw. CW_k ist offen für $k \geq 3$ bzw. $k \geq 4$. Die Klassen CW_1, CW_2, CW_3, NLC_1, NLC_2 sind in polynomieller Zeit erkennbar (Johansson, Roties).

Satz 16.5. *Sei G ein Graph mit NLC-Weite k, so daß der vollständige bipartite Graph $K_{n,n}$ mit $n > 1$ kein Teilgraph von G ist. Dann hat G höchstens die Baumweite $3\,k(n-1) - 1$.*

[13] dabei bedeutet z.B.: b_{100}^3, daß ein Knoten mit der Bezeichnung b_{100} (der Index der b-Knoten ist in Binärdarstellung) mit der Markierung 3 eingefügt wird. Ich denke, der Aufbau des Graphen ist so besser nachzuvollziehen als mit der normalen Schreibweise $\bullet_3$

Bemerkung. Viele NP-vollständige Probleme können auf Graphen mit beschränkter Cliquenweite bzw. NLC-Weite in polynomieller Zeit gelöst werden, wenn ein k-Ausruck für den Graphen gegeben ist.

16.2 Dynamische Programmierung wie bei Graphen mit beschränkter Baumweite

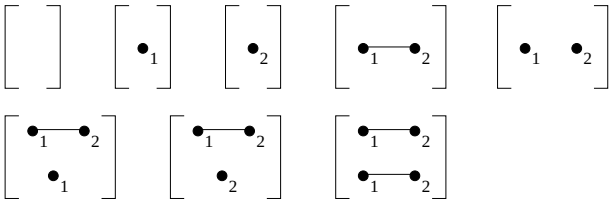
Sei $\mathcal{G}_k$ die Menge aller markierten Graphen $G = (V, E, \text{lab})$ mit $\text{lab} : V \to [k]$. Sei $\pi : \mathcal{G}_k \to \{0, 1\}$ eine Grapheigenschaft. Zwei markierte Graphen $G_1, G_2 \in \mathcal{G}_k$ sind ersettzbar bzgl. π, geschrieben $G_1 \sim_{\pi,k} G_2$, wennfür alle markierten Graphen $H \in \mathcal{G}_k$ und alle Funktionen $R : [k] \to [k]$ und alle Relationen $S \subseteq [k]^2$ gilt:

$$\pi\big(\circ_R (G_1 \times_s H)\big) = \pi\big(\circ_R (G_2 \times_s H)\big)$$

Wenn $\sim_{\pi,k}$ endlich viele Äquivalenzklassen[14] hat, dann gibt es einen Algorithmus, der $\pi(G)$ für alle $G \in \mathcal{G}_k$ in linearer Zeit entscheidet, wenn zusätzlich ein k-Ausdruck für G gegeben ist.

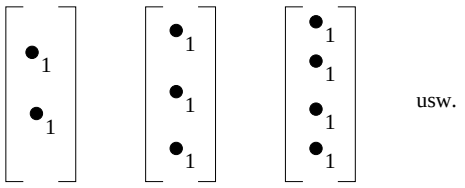
Satz 16.6. *Für alle Grapheigenschaften π, die in monadischer Logik zweiter Ordnung mit Quantifizierungen über Knoten und Knotenmengen definierbar sind, hat die Relation $\sim_{\pi,k}$ für jedes k einen endlichen Index.*[15]

Beispiel. Eigenschaft π mit endlich vielen Äquivalenzklassen für $\sim_{\pi,k}$: G ist zusammenhängend, $k = 2$



Eigenschaften π mit unendlich vielen Äquivalenzklassen für $\sim_{\pi,k}$: $(k = 1)$

1. $\pi(G) = 1$ genau dann, wenn $G = K_{n,n}$ für ein $n \geq 2$



2. $\pi(G) = 1$ genau dann, wenn G einen Hamiltonkreis hat

   Mindestens die Klassen von 1.), da der $K_{n,n}$ einen Hamiltonkreis hat.

[14] $\sim_{\pi,k}$ ist offensichtlich eine Äquivalenzrelation.

[15] Im Unterschied zu t-terminalen Graphen sind hier keine Quantifizierungen über Kanten und Kantenmengen zugelassen.

17 Der Minorensatz

Sei

$$\leq \ \subseteq X \times X$$

eine binäre Relation auf einer Menge X.

Wir schreiben $x \leq y$ anstatt $(x, y) \in \ \leq$.

Eine reflexive ($\forall x \in X : x \leq x$) und transitive ($x \leq y \wedge y \leq z \Rightarrow x \leq z$) Relation $\leq$ nennen wir eine Quasiordnung.

Sei $F = x_1, x_2, x_3, \ldots$ eine Folge von Elementen aus X.

Sei $\leq$ eine Quasiordnung auf den Elementen aus X.

Ein Paar x_i, x_j mit $i < j$ und $x_i \leq x_j$ ist ein aufsteigendes Paar in der Folge F.

Ein Paar x_i, x_j mit $i < j$ und $x_j \leq x_i$ ist ein absteigendes Paar in der Folge F.

Die Quasiordnung $\leq$ ist eine Wohlquasiordnung, wenn jede unendliche Folge ein aufsteigendes Paar enthält.

Wir definieren $x < y$, wenn $x \leq y$ und $\neg(y \leq x)$.

Bemerkung: Die Relation $<$ ist ebenfalls transitiv (Übungsaufgabe).

Ein Paar x_i, x_j mit $i < j$ und $x_i < x_j$ ist ein echt aufsteigendes Paar in der Folge F.

Ein Paar x_i, x_j mit $i < j$ und $x_i > x_j$ ist ein echt absteigendes Paar in der Folge F.

F ist eine echt aufsteigende Folge, wenn $x_i < x_{i+1}$ für $i = 1, 2, 3, \ldots$.

F ist eine echt absteigende Folge, wenn $x_i > x_{i+1}$ für $i = 1, 2, 3, \ldots$.

Eine Folge von paarweise nicht vergleichbaren Elementen ist ein Antikette.

Lemma: Eine Quasiordnung $\leq$ ist genau dann eine Wohlquasiordnung, wenn es weder eine unendliche Antikette noch eine unendliche echt absteigende Folge gibt.

Beweis:

> $\Rightarrow$ Wenn $\leq$ eine Wohlquasiordnung ist, dann enthält jede unendliche Folge ein aufsteigendes Paar (Definition der Wohlquasiordnung) und somit gibt es weder eine unendliche Antikette noch eine unendliche echt absteigende Folge.

$\Leftarrow$ Durch Widerspruch.

Angenommen es gibt weder eine unendliche Antikette noch eine unendliche echt absteigende Folge und $\leq$ ist keine Wohlquasiordnung.

Wenn $\leq$ keine Wohlquasiordnung ist, dann gibt es eine unendliche Folge F, die kein aufsteigendes Paar enthält

Betrachte nun die Teilfolge F' von F, gebildet durch die letzten Elemente aller maximalen echt absteigenden Teilfolgen in F. Diese Teilfolgen müssen nun alle endlich sein, da es laut Voraussetzung keine unendliche echt absteigende Folge gibt.

Wenn F' endlich ist, dann ist die Teilfolge von F nach dem letzten Element aus F' eine unendliche Antikette. Widerspruch!

Wenn F' unendlich ist, dann bildet F' eine unendliche Antikette. Widerspruch!

Lemma: Wenn $\leq$ eine Wohlquasiordnung ist, dann enthält jede unendliche Folge eine unendliche aufsteigende Teilfolge.

Beweis: Durch Widerspruch.

Angenommen $\leq$ ist eine Wohlquasiordnung und es gibt eine unendliche Folge F, die keine unendliche aufsteigende Teilfolge enthält.

Betrachte nun die Teilfolge F' von F, gebildet durch je ein Element aus jeder maximalen aufsteigenden Teilfolgen in F. Diese Teilfolgen sind laut Voraussetzung alle endlich.

Angenommen F' ist endlich. Dann ist die Teilfolge von F nach dem letzten Element aus F' eine unendliche Folge ohne aufsteigendes Paar und $\leq$ wäre keine Wohlquasiordnung, Widerspruch!

Angenommen F' ist unendlich. Da F' kein aufsteigendes Paar hat wäre $\leq$ keine Wohlquasiordnung. Widerspruch!

Wir betrachten nun Graphen $G = (V, E)$ mit endlichen Mengen von Knoten und Kanten.

Definition: Sei $G = (V_G, E_G)$ ein Graph und $\{u, v\} \in E$ eine Kante von G. Die Kontraktion der Kante $\{u, v\}$ erzeugt den Graphen

$$G_{v \to u} = (V', E')$$

mit

$$V' = V - \{v\}$$

und

$$E' = E - \{\{v, w\} \in E \mid w \in V\} \cup \{\{u, w\} \mid \{v, w\} \in E, w \neq u\}.$$

Ein Graph $H = (V_H, E_H)$ ist ein Minor von G, wenn H aus einem Teilgraphen G' von G durch Kantenkontraktionen gewonnen werden kann. Wir schreiben dann $H \leq G$. Die Minorenrelation $\leq$ ist reflexiv und transitiv, also eine Quasiordnung.

Satz: [Minorensatz von Robertson & Seymor] Die Menge aller Graphen ist durch die Minorenrelation $\leq$ wohlquasigeordnet.

Folgerung aus dem Minorensatz:

Eine Menge von Graphen X ist abgeschlossen bzgl. der Minorenoperation, wenn für alle Graphen $G \in X$ alle Minoren von G ebenfalls in X sind.

Satz: Für jede Menge X von Graphen, die abgeschlossen ist bzgl. der Minorenoperationen, gibt es endlich viele Graphen $H_1, \ldots, H_k$, so daß ein beliebiger Graph G genau dann nicht in X ist, wenn einer der Graphen $H_1, \ldots, H_k$ ein Minor von G ist.

Beweis: Wir konstruieren eine endliche Menge $\mathcal{H}$ von Graphen, mit folgender Eigenschaft.

$(*)$ Ein Graph G ist genau dann nicht in X, wenn einer der Graphen aus $\mathcal{H}$ ein Minor von G ist.

Initialisiere $\mathcal{H}$ als Menge aller Graphen, die nicht in X sind. Offenbar gilt die Eigenschaft (*).

Wenn $\mathcal{H}$ zwei Graphen H_1, H_2 enthält mit $H_1 \leq H_2$, dann können wir H_2 aus $\mathcal{H}$ entfernen, ohne die Eigenschaft $(*)$ zu zerstören. Wir wiederhole diese Eliminiation nun so oft wie möglich. Wenn die übrig gebliebene Menge $\mathcal{H}$ nicht endlich ist, dann gibt es eine unendliche Antikette. Widerspruch zum Minorensatz!

Bemerkungen: Für Graphen H_i fester Größe ist es in polynomieller Zeit entscheidbar, ob H_i ein Minor von G ist (geht sogar in Zeit $O((|V| + |E|)^2)$).

Folgerung: Alle Graphklassen, die abgeschlossen sind bzgl. der Minorenoperationen, sind in polynomieller Zeit erkennbar.

Bemerkungen:

- Menge der planaren Graphen
- Menger der Graphen mit beschränkter Baumweite
- ...

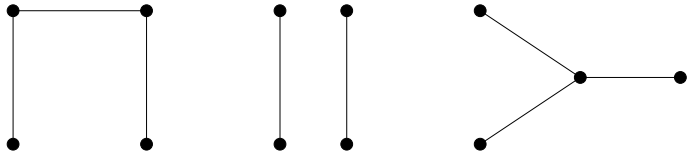
18 Extremale Graphen

Sei H ein Graph.

- Wie groß ist die maximale Anzahl von Kanten, die ein Graph mit n Knoten haben kann, so daß er keinen zu H isomorphen Teilgraphen enthält?
- Wie sieht ein solcher Graph aus?
- Ist ein solcher Graph eindeutig?
- Einen solchen Graphen nennen wir *extremal ohne Teilgraph H*.

Definition: Ein Graph $G = (V_G, E_G)$ ist genau dann *extremal ohne Teilgraph H*, wenn H kein Teilgraph von G ist, aber jeder Graph mit $|V_G|$ Knoten und mehr als $|E_G|$ Kanten enthält einen zu H isomorphen Teilgraphen.

Bemerkung: Wenn G extremal ohne Teilgraph H ist, dann ist G kantenmaximal ohne Teilgraph H. Die Umgekehrt muß das nicht gelten. Extremalität ist eine echt stärkere Eigenschaft als Kantenmaximalität.



Beispiel: Sei $H = P_4$ (links). Der Graph in der Mitte ist zwar kantenmaximal ohne den Teilgraph H, aber er ist nicht extremal, da es einen Graphen mit 3 Kanten gibt, der den Graphen H nicht enthält (rechts).
Wir betrachten extremale Graphen G ohne Teilgraph K_r, $r > 1$. Die vollständigen $(r-1)$-partiten Graphen sind kantenmaximal ohne K_r.

Die $(r-1)$-partiten Graphen, bei denen die Knoten so gleichmäßig wie möglich auf die $(r-1)$ Partitionsmengen verteilt sind [16], haben die meisten Kanten.

[16] damit ist gemeint, daß die Anzahl der Knoten in den Partitionsengen höchstens um 1 abweicht

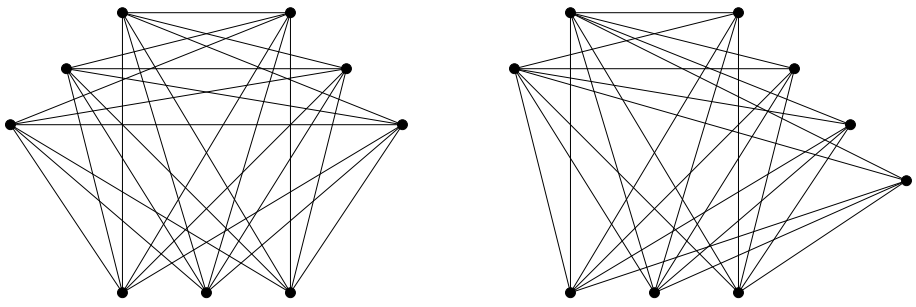


Abbildung 18.1: Der $K_{3,3,3}$ mit 27 Kanten und der $K_{2,3,4}$ mit 26 Kanten

Diese Graphen mit n Knoten nennt man Turán-Graphen, bezeichnet mit $T_{r-1}(n)$.

Satz: [Turán] Für alle $r, n \in \mathbb{N}$ mit $r > 1$ hat jeder extremale Graph G mit n Knoten ohne Teilgraph K_r genau so viele Kanten wie der $T_{r-1}(n)$.

Beweis: Wir zeigen die stärkere Aussage, dass G isomorph zu $T_{r-1}(n)$ ist

Induktion über die Anzahl der Knoten n.

Sei $G = (V_G, E_G)$ extremal ohne Teilgraph K_r. Sei n die Anzahl der Knoten in G und $t_{r-1}(n)$ die Anzahl der Kanten in $T_{r-1}(n)$.

Für $n < r$ gilt $G = K_n = T_{r-1}(n)$.

Sei also $n \geq r$. Da G kantenmaximal ohne Teilgraph K_r ist, gibt es einen Teilgraphen $H = (V_H, E_H) = K_{r-1}$ in G. Seien $x_1, \ldots, x_{r-1}$ die Knoten von H und sei $G' = (V_{G'}, E_{G'})$ der Teilgraph von G ohne die Knoten $x_1, \ldots, x_{r-1}$.

Nach Induktionsvoraussetzung hat G' höchstens $t_{r-1}\big(n-(r-1)\big)$ Kanten, und jeder Knoten in $V_{G'}$ hat in G höchstens $r-2$ Nachbarn aus V_H. Somit gilt:

$$|E_G| \leq \underbrace{t_{r-1}\big(n-(r-1)\big)}_{\text{Kanten in } G'} + \underbrace{\big(n-(r-1)\big)(r-2)}_{\text{Kanten zwischen } G' \text{ und } H} + \underbrace{\binom{r-1}{2}}_{\text{Kanten in } H}$$

Da G extremal ohne Teilgraph K_r ist und auch $T_{r-1}(n)$ keinen K_r enthält, folgt aus folgender Beobachtung sogar die Gleichheit.

Jeder Knoten in G' hat genau $r-2$ Nachbarn in H, so wie die Knoten in H selbst.

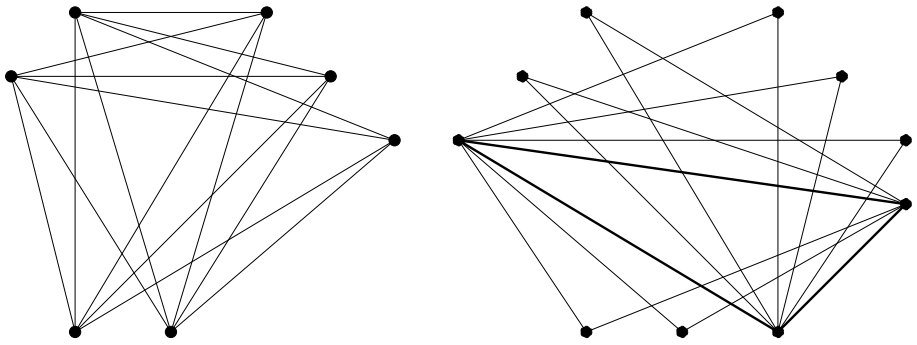
Für $i = 1, \ldots, r-1$ sei

$$V_i \;=\; \big\{ v \in V_G \;\big|\; \{v, x_j\} \in E_G \Leftrightarrow i \neq j \big\}$$

die Menge aller Knoten von G, deren $r-2$ Nachbarn in H ohne x_i sind. Da der Teilgraph K_r kein Teilgraph von G ist, sind die Knotenmengen V_i unabhängig. Somit partitionieren die V_i den Graphen G. Da der $T_{r-1}(n)$ unter allen vollständigen $(r-1)$-partiten Graphen die meisten Kanten hat, gilt $G = T_{r-1}(n)$.

Nach dem Satz von Turán hat jeder extremale Graph mit n Knoten ohne Teilgraph K_r $t_{r-1}(n) \leq \frac{1}{2} n^2 \frac{r-2}{r-1}$ Kanten. Die Gleichheit ist gegeben, wenn n durch $r-1$ teilbar ist.

Sei G ein Graph mit n Knoten, der mehr als $t_{r-1}(n)$ Kanten hat. Dann hat G nicht nur einen K_r, sondern „G ist ein Graph, in dem es von K_r-Teilgraphen nur so wimmelt“.



Sei $r = 4$ und $G = K_{3,3,4}$. G enthält einen $H = K_3$. Entfernt man diesen aus G, erhält man den linken Graphen. Nach Induktionsvoraussetzung enthält er $t_{r-1}(n - r + 1)$ Kanten. Fügt man den K_3 wieder hinzu (rechts), so muss in jeder Partitionsmenge ein Knoten hinzukommen.

Abbildung 18.2: Zum Beweis der Gleichheit.

19 Hierarchische Graphen

19.1 Definitionen und Einführung

Definition. Sei Σ ein endliches Alphabet und $G = (V_G, E_G, \mathrm{lab}_G, \Sigma)$ mit $\mathrm{lab}_G : V \to \Sigma$ ein knotenmarkierter Graph über Σ. Eine Folge P von paarweise verschiedenen Knoten aus V_G ist eine *Pinliste* für G.
Ein *Hierarchischer Graph* über Σ ist ein System

$$\Gamma = (G_1, \ldots, G_k, R, \Sigma, \delta),$$

bestehend aus einem endlichen Alphabet Σ, einem Symbol $\delta \in \Sigma$ einer Regelmenge $R : \big(\Sigma \setminus \{\delta\}\big) \to \{1, \ldots, k\}$ und $k \geq 1$ Zellen $G_1, \ldots, G_k$.
Jede Zelle $G_i = (V_i, E_i, \mathrm{lab}_i, P_i, \mathrm{num}_i)$ für $1 \leq i \leq k$ ist definiert durch

1. $(V_i, E_i, \mathrm{lab}_i, \Sigma)$ ist ein knotenmarkierter Graph über Σ,
2. P_i ist eine Pinliste für (V_i, E_i), bestehend aus p_i Knoten mit $\mathrm{lab}_i(u_j) = \delta$ für $j = 1, \ldots, p_i$,
3. $\mathrm{num}_i : E_i \to \mathbb{N}_0$ ist eine Funktion.

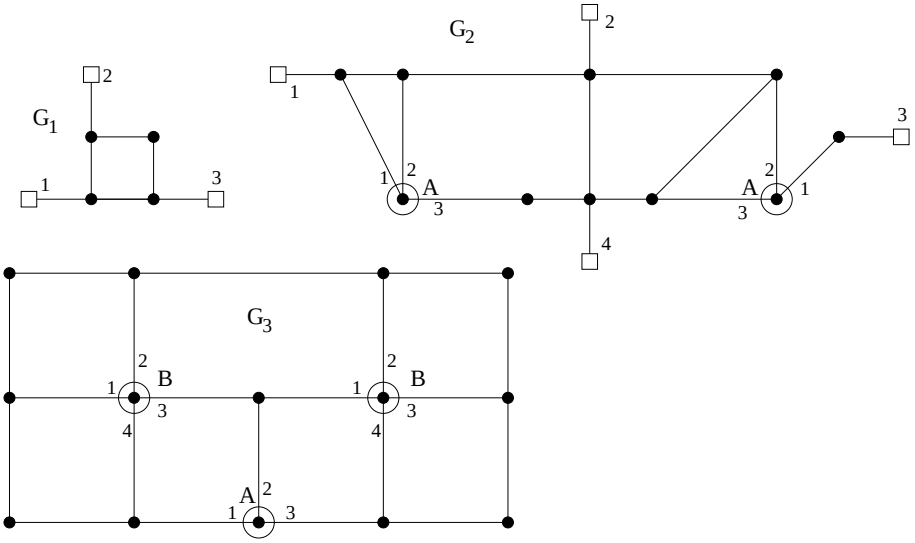
Die Symbole in $\Sigma \setminus \{\delta\}$ nennen wir *nichtterminale Symbole*, δ ist das *terminale Symbol*. Ein Knoten, der mit δ bzw. mit einem Symbol aus $\Sigma \setminus \{\delta\}$ markiert ist, heißt *terminaler* bzw. *nichtterminaler Knoten*. Nichtterminale Knoten sind nicht adjazent. Für jedes Smbol $l \in \Sigma$ gibt es mindestens einen Knoten, der mit l markiert ist. Eine Kante, die mit einem nichtterminalen Knoten inzident ist, heißt nichtterminale Kante. Alle übrigen Kanten sind terminale Kanten.
Die nichtterminalen Kanten an einem nichtterminalen Knoten sind - beginnend bei 1 - fortlaufend mit der Funktion num_i numeriert. Für terminale Kanten e gilt $\mathrm{num}_i(e) = 0$.
Ist $u \in V$ ein nichtterminaler Knoten, so hat u genau $p_{R(\mathrm{lab}_i(u))}$ Nachbarn.
Ein Ersetzungsschritt auf einer Zelle $G_i = (V_i, E_i, \mathrm{lab}_i, P_i, \mathrm{num}_i)$ und einem nichtterminalen Knoten $u \in V_i$ in einem hierarchischen Graphen $\Gamma = (G_1, \ldots, G_k, R, \Sigma, \delta)$ ist wie folgt definiert:
Sei $R\big(\mathrm{lab}_i(u)\big) = t$. Entferne den nichtterminalen Knoten u und seine inzidenten Kanten aus G_i und füge eine Kopie von G_t hinzu (disjunkte Vereinigung aller Komponenten ohne eine Erweiterung der Pinliste P_i von G_i.
War $\{u, w\}$ die j-te Kante am Knoten u, also $\mathrm{num}_i(u) = j$, und war v der j-te Knoten in Pinliste P_t, dann identifiziere v mit w. Da beides terminale Knoten sind, ist das Ergebnis wieder ein terminaler Knoten.
Die Zellen in Γ sind so geordnet, daß für alle nichtterminalen Knoten $u \in G_i$ gilt: $R\big(\mathrm{lab}_i(u)\big) < i$. Die Startzelle von Γ ist die Zelle G_k.

Die Expansion $E(\Gamma)$ ist der Graph, den man erhält, wenn man - beginnend mit der Startzelle G_k - alle nichtterminalen Knoten u der Reihe nach mit den Zellen G_t, $t = R\big(\mathrm{lab}_i(u)\big)$ ausgetauscht werden, bis keine nichtterminalen Knoten mehr übrig sind.
Die Anzahl der Knoten bzw. Kanten in Γ ist

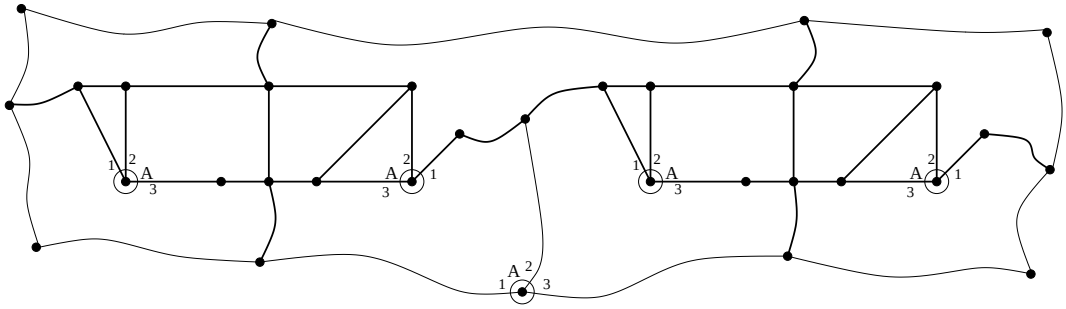
$$N_\Gamma = \sum_{i=1}^{k} |V_i| \text{ bzw. } M_\Gamma = \sum_{i=1}^{k} |E_i|.$$

Die Größe von Γ ist in $O(N_\Gamma + M_\Gamma)$, die Größe von $E(\Gamma)$ kann exponentiell in $O(N_\Gamma + M_\Gamma)$ sein.

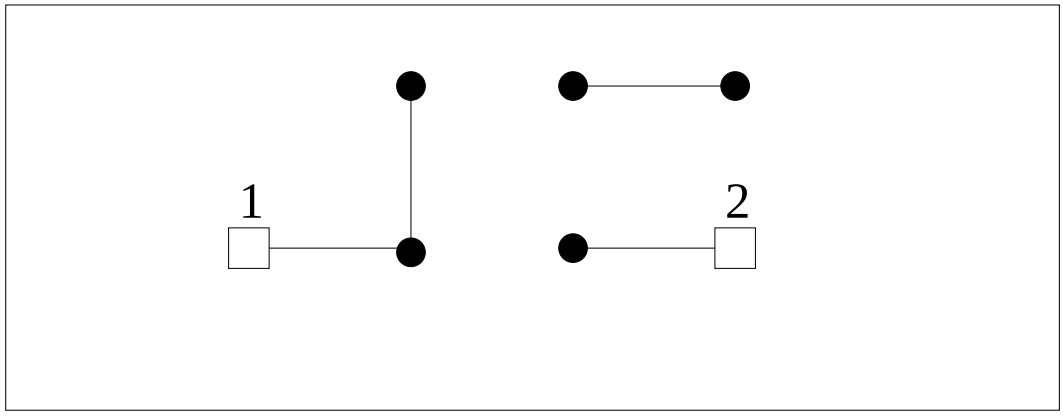
Beispiel. Sei $\Gamma = (G_1, G_2, G_3, R, \Sigma, \delta)$ mit $\Sigma = \{\delta, A, B\}$, $R(A) = 1$, $R(B) = 2$ und den drei Zellen G_1, G_2 und G_3, die wie folgt definiert sind:



Für den Aufbau der Expansion startet man mit G_3 und ersetzt in G_3 zunächst alle Knoten, die mit B markiert sind durch G_2, denn $R(B) = 2$. Man erhält folgenden Graphen



In diesem Graphen werden nun alle A-Knoten durch G_1 ersetzt, da $R(A) = 1$. Man erhält:


1
2

19.2 Algorithmen für die Analyse hierarchischer Graphen

Gegeben ist ein hierarchischer Graph Γ. Die Frage ist, ob die Expansion $E(\Gamma)$ eine bestimmte Eigenschaft erfüllt, wie zum Beispiel Zusammenhang, Planarität usw. Entscheidungsprobleme können oft mit dynamischer Programmierung effizient gelöst werden.

Definition. Sei $G_i = (V_i, E_i, \text{lab}_i, P_i, \text{num}_i)$ eine Zelle mit m nichtterminalen Knoten $u_1, \ldots, u_m$. Seien $H_1, \ldots H_m$ h_i-terminale Graphen mit $h_i = |N(u_i)|$. Dann ist der t-terminale Graph

$$G_i\big[u_1/H_1, \ldots, u_m/H_m\big]$$

mit $t = |P_i|$ wie folgt definiert:

Falls $m = 0$ $G_i[] = (V_i, E_i, P_i)$

Falls $m > 0$ Ersetze jeden nichtterminalen Knoten u_j in G_i für $j = 1, \ldots, m$ durch eine Kopie von H_j und identfiziere den l-ten Knoten aus der Pinliste der Kopie mit dem l-ten Knoten in der Nachbarschaft von u_j [17].

Das Ergebnis ohne die Funktion num_i und lab_i ist der t-terminale Graph $G_i\big[u_1/H_1, \ldots, u_m/H_m\big]$

Die Expansion $E(\Gamma)$ kann wie folgt konstruiert werden: Sei $\Gamma = (G_1, \ldots, G_k, R, \Sigma, \delta)$.

```
for i = 1,...,k do {
    seien u_1,...,u_m die nichtterminalen Knoten in G_i;
    if m = 0 then G̃_i = G_i[ ];
    if m > 0 then G̃_i = G_i[u_1/G̃_{R(lab_i(u_1))},...,u_m/G̃_{R(lab_i(u_m))}];
}
```

Das Ergebnis $\tilde{G}_k$ ist $E(\Gamma)$.

Definition. Zwei t-terminale Graphen G_1 und G_2 sind ersetzbar bzgl. der Eigenschaft Π (in Zeichen $G_1 \sim_{\Pi,t} G_2$), wenn für alle t-terminalen Graphen H gilt:

$$\Pi(G_1 \circ H) = \Pi(G_2 \circ H)$$

Für einen t-terminalen Graphen G und eine Eigenschaft Π sei $G^b = \text{burn}(G)$ ein kleiner t-terminaler Graph mit $G^b \sim_{\Pi,t} G$.[18]

[17] der l-te Knoten in der Nachbarschaft von u ist durch die Funktion $\text{num}_i(u)$ gegeben

[18] Der Graph G wird „klein gebrannt“.

19.2.1 Algorithmisches Gerüst für die Analyse von hierarchischen Graphen

`Bottom-Up-Prozedur`
`for` $i = 1, \ldots, k$ `do` $\{$
`seien` $u_1, \ldots, u_m$ `die nichtterminalen Knoten in` G_i;
`if` $m = 0$ `then` $\tilde{G}_i = G_i[\quad]$;
`if` $m > 0$ `then` $\tilde{G}_i = G_i\big[u_1/\tilde{G}^b_{R(\mathtt{lab}_i(u_1))}, \ldots, u_m/\tilde{G}^b_{R(\mathtt{lab}_i(u_m))}\big]$;
$G^b_i = \mathtt{burn}(\tilde{G}_i);$ $\}$

Die Effizienz der Bottom-Up-Prozedur hängt natürlich von der Güte der Brennprozedur burn() ab.

19.2.2 Brennprozedur burn() für Π =Zusammenhang

Sei $G = (V_G, E_G, P_G)$ ein t-terminaler Graph und $J = (V_J, E_J, P_J)$ ein t-terminaler Graph mit $|V_J| = |P_J| = |P_G|$ und $|E_J| = 0$.
Berechne die Zusammenhangskomponenten von G. Wenn G eine Komponente enthält, die keinen Knoten aus P_G enthält, dann füge einen isolierten Knoten u zu J hinzu.
Für jede Zusammenhangskomponente Z_i in G mit mindestens einem Knoten aus P_G füge einen neuen Knoten u_i zu J hinzu und verbinde u_i genau dann mit dem j-ten Knoten aus P_J, wenn der j-te Knoten aus P_G in Z_i enthalten ist.
Sonderfall: Wenn die Pinliste leer und der Graph nicht zusammenhängend ist, dann besteht J aus zwei isolierten Knoten.
Der daraus entstehende Graph J hat $O(|P_J|)$ Knoten und Kanten, kann in Zeit $O(|V_G| + |E_G|)$ berechnet werden und ist ersetzbar mit G bzgl. Zusammenhang.
Wird dieser Brennalgorithmus in der Bottom-Up-Prozedur eingesetzt, dann ist die Anzahl der Knoten und Kanten in $\tilde{G}_i$ linear in der Anzahl der Knoten und Kanten in G_i.

Satz 19.1. *Sei $\Gamma = (G_1, \ldots, G_k, \Sigma, R, \delta)$ ein hierarchischer Graph. Die Frage, ob die Expansion $E(\Gamma)$ zusammenhängend ist, kann in Zeit $O(N_\Gamma + M_\Gamma)$ entschieden werden.*

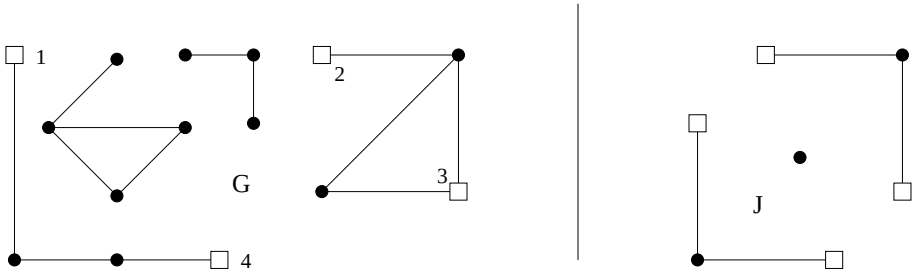


Abbildung 19.1: Beispiel für die burn-Prozedur für Π =Zusammenhang

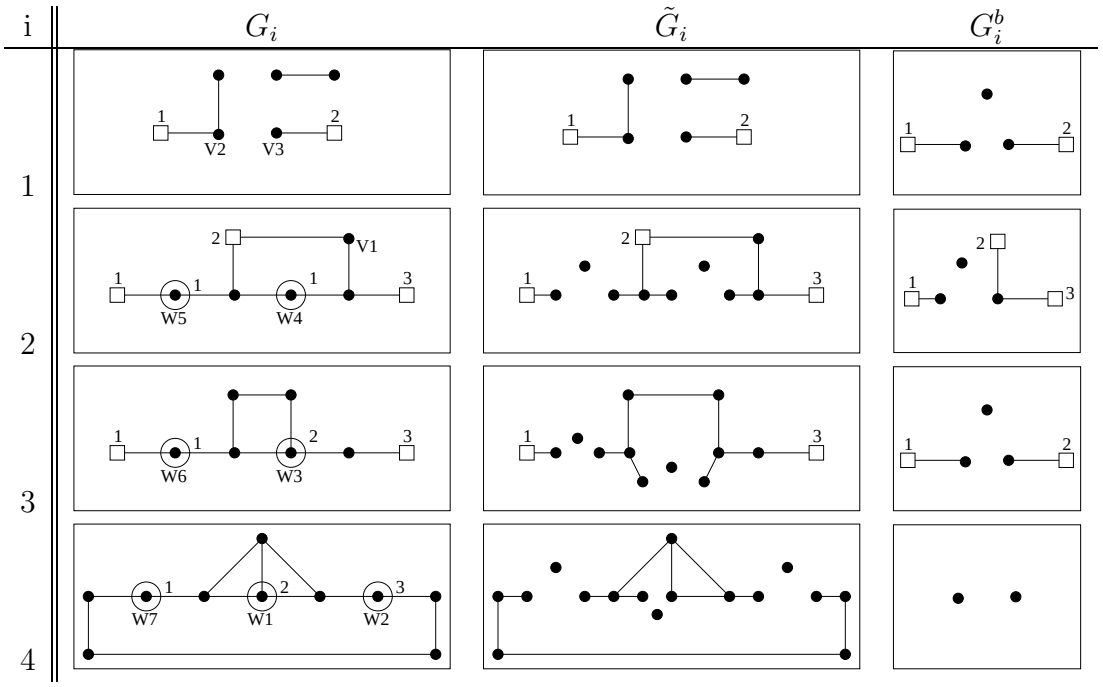


Abbildung 19.2: Bottom-Up-Tabelle für einen hierarchischen Graphen Γ

In Abbildung 19.2 ist ein komplettes Beispiel vorgeführt.

Definition. Der *Hierarchiebaum* $T(\Gamma)$ für $\Gamma = (G_1, \ldots, G_k, \Sigma, R, \delta)$ ist ein gerichteter Baum mit einer Wurzel, die mit k markiert ist. Jeder Knoten in $T(\Gamma)$, der mit $1 \leq l \leq k$ markiert ist, hat einen Sohn für jeden nichtterminalen Knoten u_i in Zelle G_l, der mit $R(\text{lab}_l(u_i))$ markiert ist.

Knoten in der Expansion können durch Pfadnamen spezifiziert werden. Ein Pfadname ist ein Weg in $T(\Gamma)$ von der Wurzel zu einem mit l markierten Knoten u, gefolgt von einm terminalen Knoten aus G_l. Betrachte dazu die Abbildungen 19.2, 19.3 und 19.4

19.2.3 Anfrageprobleme

Gegeben sind zwei Pfadnamen für zwei Knoten in $E(\Gamma)$. Die Frage, ob die beiden Knoten in derselben Zusammenhangskomponente liegen, kann mit dem obigen

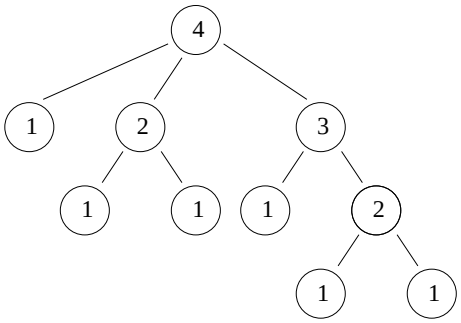


Abbildung 19.3: Der Hierarchiebaum für Γ aus Abbildung 19.2

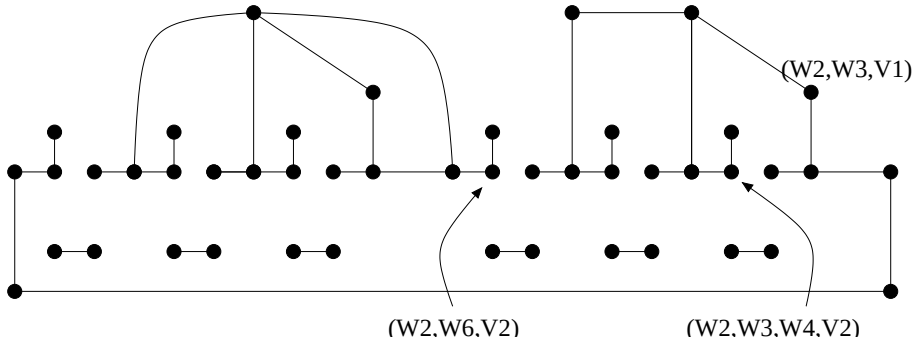


Abbildung 19.4: Die Expansion für Γ aus Abbildung 19.2 mit einigen Pfadnamen

Verfahren in Linearzeit beantwortet werden.

19.2.4 Konstruktionsprobleme

Ziel ist es, für jede Zusammenhangskomponente der Expansion eine hierarchische Beschreibung zu finden. Zerlege dazu Γ wie folgt:

- Erzeuge neue Zellen $G_{i,j}$ für jede Zusammenhangskomponente Z_j in $\tilde{G}_i$, die Pins enthält. Alle Komponenten ohne Pins werden zu einer Komponente zusammengefasst.

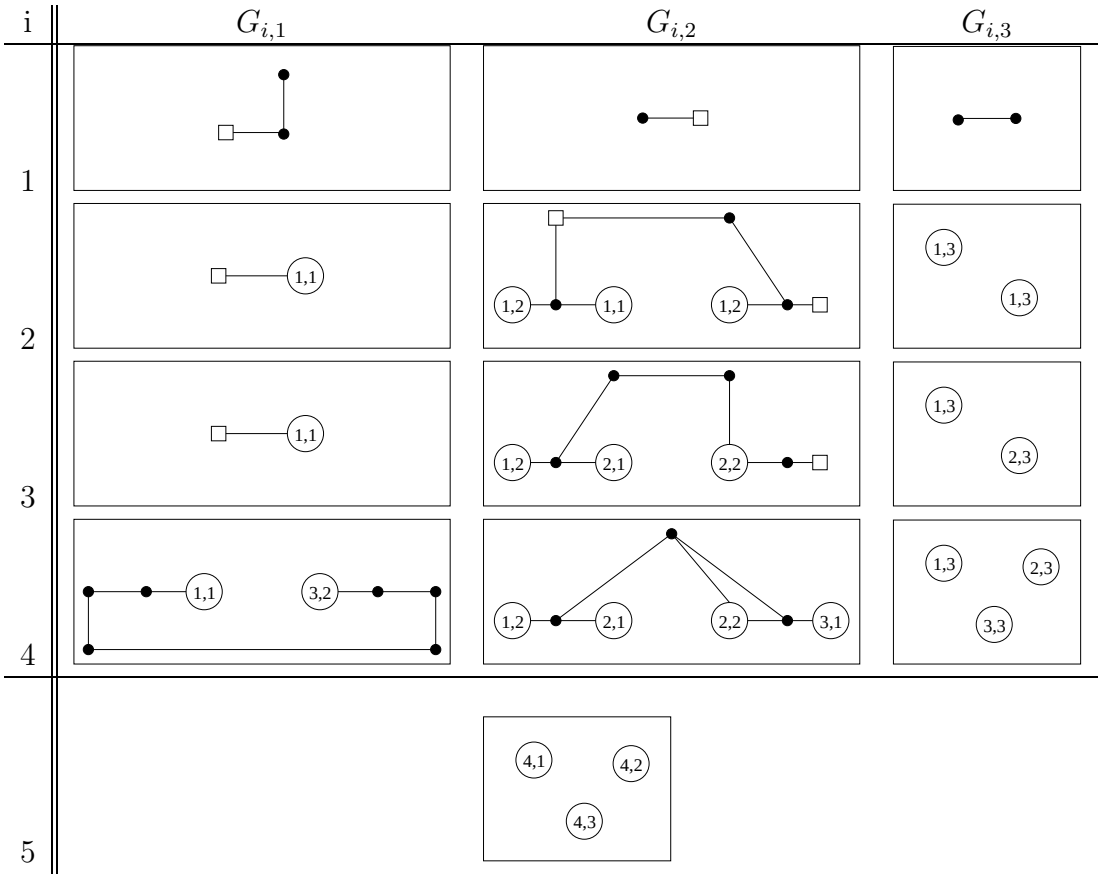
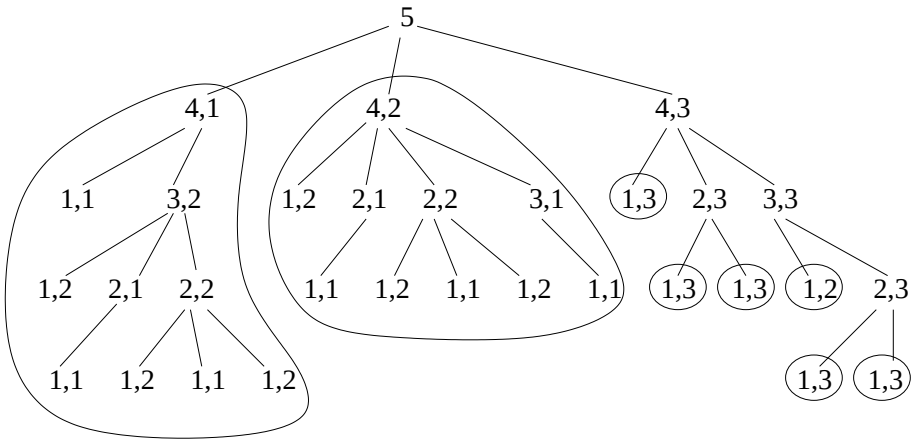


Abbildung 19.5: hierarchische Beschreibung der Zusammenhangskomponenten des Graphen aus Abbildung 19.2

- Ersetze in Zelle G_t, $t > i$ die nichtterminalen u mit $R(\text{lab}_t(u)) = i$ durch mehrere nichtterminale Knoten für die zusammenhängende Komponente in $\tilde{G}_i$. Alle nichtterminalen Knoten, die eine Zusammenhangskomponente ohne Pins repräsentieren, werden durch einen gemeinsamen Knoten dargestellt. Enthält die Startzelle mehrere Zusammenhangskomponenten, dann wird eine neue Wurzelzelle erzeugt.

In Abbildung 19.5 wird das Verfahren anhand des Graphen aus dem letzten Beispiel vorgeführt. Dabei stehen die Zahlen in den Kreisen für den Index der Zelle, die an dieser Stelle eingesetzt wird.
Natürlich kann man auch hierfür wieder den Hierarchiebaum erstellen:



Satz 19.2. *Sei Γ' das Ergebnis der obigen Zerlegung. Dann gilt:*

1. $N_{\Gamma'}, M_{\Gamma'} \in O(N_\Gamma + M_\Gamma)$
2. $E(\Gamma') = E(\Gamma)$
3. *Jede Zusammenhangskomponente von $E(\Gamma)$ ist durch eine Auswahl der Zellen aus Γ' darstellbar.*

19.2.5 Optimierungsprobleme

Beispiel. Gegeben ist ein hierarchischer Graph Γ.
Frage: Wie groß sind die Kosten eines minimalen Spannwaldes von $E(\Gamma)$?

Erweiterung der Ersetzbarkeit auf Optimierungsprobleme
Für einen Graphen G sei $\mathcal{P}(G)$ die Menge aller Teilgraphen von G. Für ein Optimierungsproblem Π sei $S(G) \subseteq \mathcal{P}(G)$ die Menge der Lösungen für G. Sei $f : \mathcal{G} \to \mathbb{R}$ eine Kostenfunktion für die Lösungen. Erweiterung der Ersetzbarkeit: Zwei t-terminale Graphen G_1, G_2 sind ersetzbar bzgl. Π mit Kostenfunktion f, $G_1 \sim_{\Pi,t} G_2$, wenn für alle t-terminale Graphen H und alle Werte $k \in \mathbb{R}$ gilt:
$G_1 \circ H$ hat einen Teilgraphen J_1 mit $f(J_1) \leq k$ (bzw. $f(J_1) \geq k$) genau dann, wenn $G_2 \circ H$ einen Teilgraphen J_1 mit $f(J_2) \leq k$ (bzw. $f(J_2) \geq k$) hat.

Brennprozedur für minimale Spannwälder
Sei $G = (V_G, E_G, P_G, f)$ ein t-terminaler Graph mit Kantengewichtsfunktion $f : E_G \to \mathbb{R}$. Die Kosten eines Teilgraphen J von G ist die Summe der Kantengewichte in J.

Schritt 1. Berechne einen minimalen Spannwald G_1 für G, indem zum Beispiel aus jedem Kreis in G eine Kante mit maximalen Gewicht entfernt wird. [19] Offensichtlich gilt: $G_1 \sim_{\Pi,t} G$, da für alle H die Kreise in G auch Kreise in $G \circ H$ sind.

Schritt 2. Initialisiere eine Variable $x = 0$;

Schritt 3. Sei u ein Knoten vom Grad 1 in G_1, der nicht in der Pinliste ist und sei w der Nachbar von u. Entferne u und seine inzidente Kante $e = \{u, w\}$ und setze $x = x + f(e)$.

Schritt 4. Sei u ein Knoten vom Grad 2, der nicht in der Pinliste ist und seien w_1 und w_2 die beiden Nachbarn von u. Entferne u und seine inzidenten Kanten $e_1\{u, w_1\}$ und $e_2\{u, w_2\}$ aus G_1 und füge eine neue Kante $e = \{w_1, w_2\}$ mit $f(e) = \max\{f(e_1), f(e_2)\}$ hinzu. Setze $x = x + \min\{f(e_1), f(e_2)\}$.

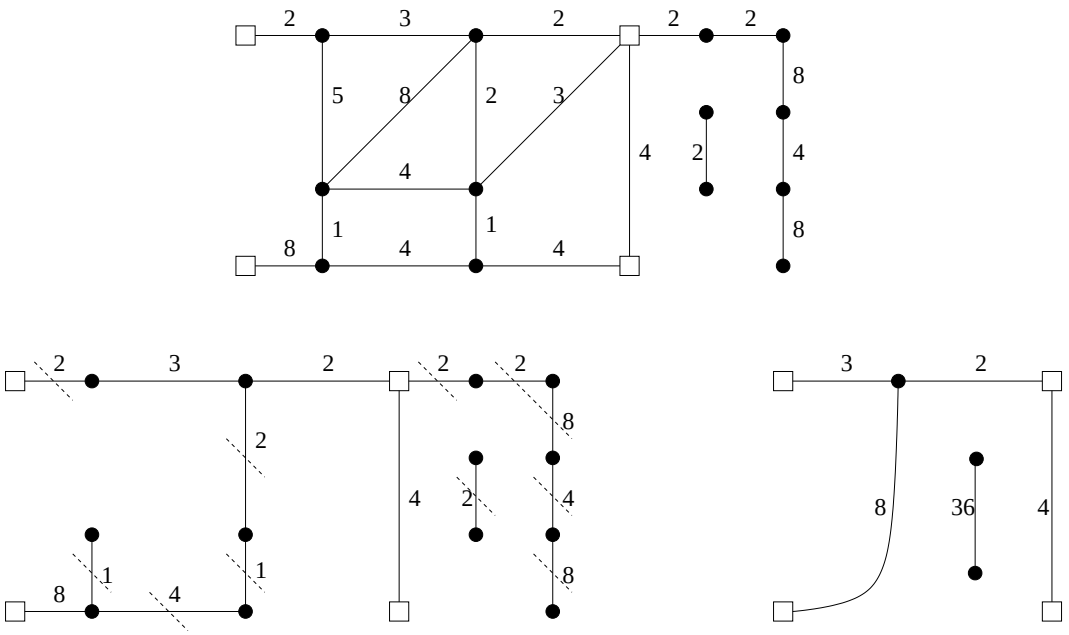
Schritt 5. Füge zum Schluss zwei neue Knoten v_1, v_2 und eine neue Kante $\{v_1, v_2\}$ mit Gewicht $f(\{v_1, v_2\}) = x$ hinzu.

Schritt 6. Lösche alle Knoten vom Grad 0.

Das Ergebnis G^b ist ersetzbar mit G_1 bzgl. der Kosten eines minimalen Spannwaldes und somit $G^b \sim_{\Pi,t} G$.
G^b kann in Zeit $= O\big(|V_G| + |E_G| + h(|V_G|, |E_G|)\big)$ berechnet werden, wenn ein minimaler Spannwald für G in Zeit $h(|V_G|, |E_G|)$ berechnet werden kann.
G^b hat $O(t) = O(|P|)$ viele Knoten und Kanten.

Satz 19.3. *Die Kosten eines minimalen Spannwaldes für die Expansion eines hierarchischen Graphen* Γ *kann in polynomieller Zeit* $O\big(h(|N_\Gamma|, |M_\Gamma|)\big)$ *berechnet werden.*

[19] Erinnerung: Rote Regel für minimale Spannwälder: Aus jedem Kreis kann eine beliebige Kante mit maximalen Gewicht entfernt werden. Das Ergebnis hat genau dann einen minimalen Spannwald mit Kosten k, wenn der Originalgraph einen minimalen Spannwald mit Kosten k hat.



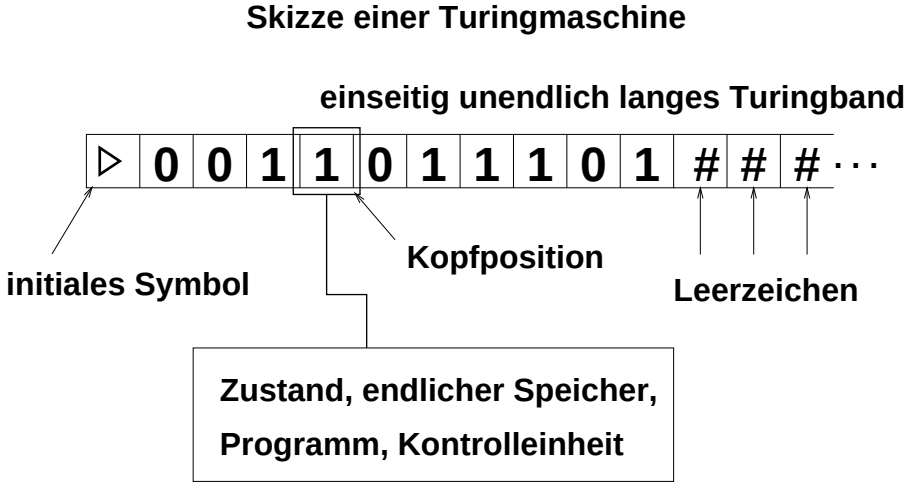
Oben: Der Graph G
Unten links: Ein minimaler Spannbaum G_1 für G. Die Kanten, die in den Schritten 3 und 4 entfernt werden, sind markiert. Rechts daneben G^b

Abbildung 19.6: Beispiel für die Brennprozedur für minimale Spannwälder

20 NP-Vollständigkeit - Theorie und Anwendungen

20.1 Turingmaschinen

Eine Turingmaschine ist ein einfaches, formales, mathematisches Modell, mit dem die Berechenbarkeit eines Allzweckcomputers modelliert werden kann. Die Turingmaschine wurde 1936 von Alan Turing entwickelt.



Definition: Eine *Turingmaschine* ist ein System $M = (Q, \Sigma, \delta, s)$ mit:

- Q ist eine endliche Menge von *Zuständen*,
- $s \in Q$ ist der *Startzustand*,
- Σ ist eine endliche Menge von *Symbolen* (das *Alphabet* von M), die immer die beiden Symbole # (*Leerzeichen*) und $\triangleright$ (*initiales Symbol*) enthält, und
- $\delta : Q \times \Sigma \longrightarrow (Q \cup \{h, h_y, h_n\}) \times \Sigma \times \{\rightarrow, \leftarrow, \perp\}$ ist die *Übergangsfunktion* (das *Programm*) mit den *Haltezuständen*

  h *allgemeiner* Haltezustand,

  h_y *akzeptierender* Haltezustand und

  h_n *ablehnender* Haltezustand,

  und den *Richtungen*

  $\rightarrow$ *rechts*,

  $\leftarrow$ *links* und

  $\perp$ *stehenbleiben*.

Motivation:

Eine Turingmaschine M arbeitet mit einem *Kopf* auf einem *Band* (*Turingband*). Das Turingband ist zu Beginn jeder Berechnung mit dem Wort "$\triangleright w$" initialisiert, wobei $w \in (\Sigma - \{\#, \triangleright\})^*$ die *Eingabe* für M ist. Der Kopf steht zu Beginn jeder Berechnung auf dem ersten (initialen) Symbol $\triangleright$. In den Beispielen kennzeichnen wir die Kopfposition durch ein unterstrichenes Symbol.

Einschränkung:

$\forall\ q \in Q\ :\ \delta(q, \triangleright) = (q', \triangleright, \rightarrow)$ oder $(q', \triangleright, \perp)$ für ein $q' \in Q$, d.h., M läuft niemals links über das initiale Symbol $\triangleright$ hinweg. Das initiale Symbol wird niemals überschrieben.

Definition: Eine *Konfiguration* einer Turingmaschine $M = (Q, \Sigma, \delta, s)$ ist ein Tripel (q, u, w) mit:

- $q \in Q \cup \{h, h_y, h_n\}$ ist der *aktuelle Zustand*,
- $u \in \{\triangleright\} \times \Sigma^*$ ist das *Wort links vom Kopf* einschließlich dem Symbol, auf dem der Kopf steht, und
- $w \in \Sigma^*$ ist das *Wort rechts vom Kopf*, ohne die hinteren Leerzeichen, die nicht explizit von M auf das Band geschrieben wurden.

Definition: Konfiguration (q', u', w') ist eine Nachfolgekonfiguration von Konfiguration (q, u, w) mit $u = \hat{u}a$, $u \in \Sigma^*$, $a \in \Sigma$, bezeichnet mit $(q, u, w) \overset{M}{\longmapsto} (q', u', w')$, falls $\delta(q, a) = (q', b, D)$ für ein $b \in \Sigma$ und ein $D\{\rightarrow, \leftarrow, \perp\}$, so dass folgende Bedingungen erfüllt sind:

- Falls $D = \perp$, dann ist $u' = \hat{u}b$ und $w' = w$,
- falls $D = \leftarrow$, dann ist $u' = \hat{u}$ und $w' = bw$,
- falls $D = \rightarrow$ und $w = \epsilon$, dann ist $u' = \hat{u}b\#$ und $w' = \epsilon$,
- falls $D = \rightarrow$ und $w = c\hat{w}$ für ein $c \in \Sigma$ und $\hat{w} \in \Sigma^*$, dann ist $u' = \hat{u}bc$ und $w' = \hat{w}$.

$\overset{M^k}{\longmapsto}$ bezeichnet eine Berechnung in k Schritten.

$\overset{M^*}{\longmapsto}$ bezeichnet eine Berechnung in endlich vielen Schritten (= reflexive und transitive Hülle von $\overset{M}{\longmapsto}$).

Die *Initiale Konfiguration* für eine Turingmaschine $M = (Q, \Sigma, \delta, s)$ mit Eingabe $x \in \Sigma - \{\triangleright, \#\}$ ist $\beta_0 = (s, \triangleright, x)$.

Eine Konfiguration $\beta = (q, u, w)$ mit $q \in \{h, h_y, h_n\}$, heißt *Haltekonfiguration*.

Wir sagen, M *hält auf Eingabe* x, falls es eine Berechnung $(s, \triangleright, x) \overset{M^*}{\longmapsto} (q, w, u)$ gibt mit $q \in \{h, h_y, h_n\}$.

Definition: Die *Ausgabe* einer Turingmaschine, bezeichnet mit $M(x)$, ist wie folgt definiert.

- Falls M auf Eingabe x im Zustand h hält, also eine Berechnung $(s, \triangleright, x) \overset{M^*}{\longmapsto} (h, w, u)$ existiert, so ist $M(x)$ das Wort, das auf dem Band rechts vom ersten initialen Symbol steht ohne die hinteren Leerzeichen, die nicht explizit von M auf das Band geschrieben wurden, also $M(x) = w'u$ mit $w = \triangleright w'$.
- Falls M auf Eingabe x im Zustand h_y oder h_n hält, so ist $M(x) =$„yes" bzw. $M(x) =$„no".
- Falls M auf Eingabe x nicht hält, es also keine Berechnung $(s, \triangleright, x) \overset{M^*}{\longmapsto} (q, w, u)$ mit $q \in \{h, h_y, h_n\}$ existiert, so ist $M(x) =$„$\nearrow$".

Beispiel:

Sei $M = (Q, \Sigma, \delta, s)$ mit $Q = \{s, q_0, q_1, q\}$, $\Sigma = \{0, 1, \triangleright, \#\}$, und

δ	0	1	#	$\triangleright$
s	$(s, 0, \rightarrow)$	$(s, 1, \rightarrow)$	$(q, \#, \leftarrow)$	$(s, \triangleright, \rightarrow)$
q_0	$(s, 0, \leftarrow)$	$(s, 0, \leftarrow)$	$(s, 0, \leftarrow)$	$(h, \triangleright, \perp)$
q_1	$(s, 1, \leftarrow)$	$(s, 1, \leftarrow)$	$(s, 1, \leftarrow)$	$(h, \triangleright, \perp)$
q	$(q_0, \#, \rightarrow)$	$(q_1, \#, \rightarrow)$	$(q, \#, \perp)$	$(h, \triangleright, \perp)$

Beispiel einer Berechnung:

s,	$\underline{\triangleright}$	0	1	0	
s,	$\triangleright$	$\underline{0}$	1	0	
s,	$\triangleright$	0	$\underline{1}$	0	
s,	$\triangleright$	0	1	$\underline{0}$	
s,	$\triangleright$	0	1	0	$\underline{\#}$
q,	$\triangleright$	0	1	$\underline{0}$	#
q_0,	$\triangleright$	0	1	#	$\underline{\#}$
s,	$\triangleright$	0	1	$\underline{\#}$	0

q,	$\triangleright$	0	$\underline{1}$	#	0
q_1,	$\triangleright$	0	#	$\underline{\#}$	0
s,	$\triangleright$	0	$\underline{\#}$	1	0
q,	$\triangleright$	$\underline{0}$	#	1	0
q_0,	$\triangleright$	#	$\underline{\#}$	1	0
s,	$\triangleright$	$\underline{\#}$	0	1	0
q,	$\underline{\triangleright}$	#	0	1	0
h,	$\underline{\triangleright}$	#	0	1	0

Definition: Sei $L \subseteq (\Sigma - \{\#, \triangleright\})^*$ eine Sprache und $M = (Q, \Sigma, \delta, s)$ eine Turingmaschine.

- M *entscheidet* L, falls $\forall\ x \in (\Sigma - \{\#, \triangleright\})^*$:
  $x \in L \ \Rightarrow\ M(x) =$"yes" und $x \notin L \ \Rightarrow\ M(x) =$"no".
- M *akzeptiert* L, falls $\forall\ x \in (\Sigma - \{\#, \triangleright\})^*$:
  $x \in L \ \Rightarrow\ M(x) =$"yes" und $x \notin L \ \Rightarrow\ M(x) =$„↗".
- L heißt *rekursiv*, wenn es eine Turingmaschine M gibt, die L entscheidet.
- L heißt *rekursiv aufzählbar*, wenn es eine Turingmaschine M gibt, die L akzeptiert.

Satz: Jede rekursive Sprache ist rekursiv aufzählbar.

(Die Umkehrung gilt jedoch nicht!)

Beweis: Sei $M = (Q, \Sigma, \delta, s)$ eine Turingmaschine, die L entscheidet.

Ändere alle Programmschritte $\delta(q, a) = (h_n, w, D)$ für $q \in Q, a \in \Sigma, w \in \Sigma$ und $D \in \{\leftarrow, \rightarrow, \perp\}$ in $\delta(q, a) = (q, a, \perp)$.

Wurde vorher der Zustand h_n erreicht, so läuft M nun in eine „Endlosschleife". □

Definition: Sei $f : (\Sigma - \{\#, \triangleright\})^* \longrightarrow \Sigma^*$ eine (totale) Funktion und M eine Turingmaschine mit Alphabet Σ.

- M berechnet f, falls $\forall\ x \in (\Sigma - \{\#, \triangleright\})^*\ :\ M(x) = f(x)$.
- f ist eine *rekursive Funktion*, falls es eine Maschine M mit $M(x) = f(x)$ gibt.

Sei $f : (\Sigma - \{\#, \triangleright\})^* \longrightarrow \Sigma^*$ eine partielle Funktion und M eine Turingmaschine mit Alphabet Σ.

- M berechnet f, falls $\forall\ x \in (\Sigma - \{\#, \triangleright\})^*$ mit definiertem $f(x) : M(x) = f(x)$.
- f ist eine *partiell rekursive Funktion*, falls es eine Maschine M gibt, die f berechnet.

Beispiele:

$f : \{0, 1\}^* \longrightarrow \{0, 1, \#, \triangleright\}^*$ mit $f(x) = \#x$ ist eine rekurisve Funktion, siehe obiges Beispiel.

Eine Turingmaschine für Palindrome:

Sei $M = (Q, \Sigma, \delta, s)$ mit $Q = \{s, q_0, q_1, q, q_0^1, q_1^1\}$, $\Sigma = \{0, 1, \triangleright, \#\}$ und

	0	1	$\triangleright$	$\#$
s	$(q_0, \triangleright, \rightarrow)$	$(q_1, \triangleright, \rightarrow)$	$(s, \triangleright, \rightarrow)$	$(h_y, \#, \perp)$
q_0	$(q_0, 0, \rightarrow)$	$(q_0, 1, \rightarrow)$	$-$	$(q_0^1, \#, \leftarrow)$
q_1	$(q_1, 0, \rightarrow)$	$(q_1, 1, \rightarrow)$	$-$	$(q_1^1, \#, \leftarrow)$
q_0^1	$(q, \#, \leftarrow)$	$(h_n, 1, \perp)$	$(h_y, \triangleright\perp)$	$-$
q_1^1	$(h_n, 1, \perp)$	$(q, \#, \leftarrow)$	$(h_y, \triangleright, \perp)$	$-$
q	$(q, 0, \leftarrow)$	$(q, 1, \leftarrow)$	$(s, \triangleright, \rightarrow)$	$-$

Die '$-$' Einträge können beliebig gewählt werden.

Obige Maschine M entscheidet die Menge aller Palindrome über $\{0, 1\}$.

Church'sche Hypothese:

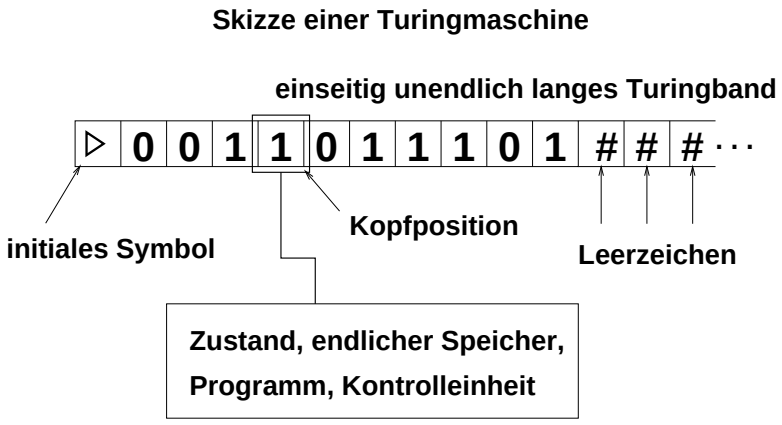
Die „intuitiv berechenbaren" Funktionen sind genau die partiell rekursiven Funktionen.

Einen Beweis kann es nicht geben, solange der Begriff „intuitiv berechenbar" informell bleibt.

Definition: Eine *k-Band-Turingmaschine* ist eine System $M = (Q, \Sigma, \delta, s)$, wobei Q, Σ und s wie für eine 1-Band-Turingmaschine definiert sind und

$$\delta : Q \times \Sigma^k \longrightarrow (Q \cup \{h, h_y, h_n\}) \times (\Sigma \times \{\rightarrow, \leftarrow, \perp\})^k$$

das Programm von M ist.



Unterschiede zur 1-Band-Turingmaschine:

- Eine k-Band-Turingmaschine hat k Bänder und k Köpfe.

- Sei $\delta(q, a_1, \ldots, a_k) = (q', b_1, D_1, \ldots, b_k, D_k)$ für
  $q \in Q, a_1, \ldots, a_k \in \Sigma, b_1, \ldots, b_k \in \Sigma$ und $D_1, \ldots, D_k \in \{\rightarrow, \leftarrow, \perp\}$.

  Wenn M im Zustand q ist und Kopf i, $1 \leq i \leq k$, auf Band i das Symbol a_i liest, wechselt die Maschine in den Zustand q' und überschreibt a_i mit b_i und bewegt Kopf i in Richtung D_i.

- Die Eingabe steht auf dem ersten Band, die Ausgabe im Zustand h steht auf dem letzten Band.

- Ein Kopf bewegt sich niemals nach links über das initiale Symbol hinaus. Die initialen Symbole werden niemals überschrieben.

- Konfigurationen sind $(1 + 2 \cdot k)$-Tupel.

- Eine k-Band-Turingmaschine startet in der Konfiguration (*Startkonfiguration*)
$$\beta_0 = (s, \triangleright, x, \triangleright, \varepsilon, \cdots \triangleright, \varepsilon).$$

- Die Relationen $\overset{M}{\longmapsto}$, $\overset{M^k}{\longmapsto}$, $\overset{M^*}{\longmapsto}$ sind analog zur 1-Band-Turingmaschinen definiert. (Übungsaufgabe!)

- $M(x) =$„yes", falls $\beta_0 \overset{M^*}{\longmapsto} (h_y, w_1, u_1, \ldots, w_k, u_k)$ für $w_1, u_1, \ldots, w_k, u_k \in \Sigma^*$.

- $M(x) =$„no", falls $\beta_0 \overset{M^*}{\longmapsto} (h_n, w_1, u_1, \ldots, w_k, u_k)$ für $w_1, u_1, \ldots, w_k, u_k \in \Sigma^*$.

- $M(x) = y \in \Sigma^*$, falls $\beta_0 \overset{M^*}{\longmapsto} (h, w_1, u_1, \ldots, w_k, u_k)$ für $w_1, u_1, \ldots, w_k, u_k \in \Sigma^*$ und y das Wort $w_k u_k$ ist ohne führendes initiales Symbol und ohne nachfolgende Leerzeichen.

- $M(x) =$„$\nearrow$", falls die Maschine auf Eingabe x nicht hält.

Beispiel:
Sei $M = (Q, \Sigma, \delta, s)$, $Q = \{s, p, q\}$, und $\Sigma = \{0, 1, \triangleright, \#\}$

q	a_1	a_2	$\delta(q, a_1, a_2)$		
s	0	$\#$	s	$(0, \rightarrow)$	$(0, \rightarrow)$
s	1	$\#$	s	$(1, \rightarrow)$	$(1, \rightarrow)$
s	$\triangleright$	$\triangleright$	s	$(\triangleright, \rightarrow)$	$(\triangleright, \rightarrow)$
s	$\#$	$\#$	q	$(\#, \leftarrow)$	$(\#, \perp)$
q	0	$\#$	q	$(0, \leftarrow)$	$(\#, \perp)$
q	1	$\#$	q	$(1, \leftarrow)$	$(\#, \perp)$
q	$\triangleright$	$\#$	p	$(\triangleright, \rightarrow)$	$(\#, \leftarrow)$
p	0	0	p	$(0, \rightarrow)$	$(0, \leftarrow)$
p	0	1	h_n	$(0, \perp)$	$(1, \perp)$
p	1	0	h_n	$(1, \perp)$	$(0, \perp)$
p	1	1	p	$(1, \rightarrow)$	$(1, \leftarrow)$
p	$\#$	$\triangleright$	h_y	$(\#, \perp)$	$(\triangleright, \perp)$

	Band 1							Band 2						
s	$\underline{\triangleright}$	0	1	0	1	0		$\underline{\triangleright}$	$\#$	$\#$	$\#$	$\#$	$\#$	
s	$\triangleright$	$\underline{0}$	1	0	1	0		$\triangleright$	$\underline{\#}$	$\#$	$\#$	$\#$	$\#$	
s	$\triangleright$	0	$\underline{1}$	0	1	0		$\triangleright$	0	$\underline{\#}$	$\#$	$\#$	$\#$	
s	$\triangleright$	0	1	$\underline{0}$	1	0		$\triangleright$	0	1	$\underline{\#}$	$\#$	$\#$	
s	$\triangleright$	0	1	0	$\underline{1}$	0		$\triangleright$	0	1	0	$\underline{\#}$	$\#$	
s	$\triangleright$	0	1	0	1	$\underline{0}$		$\triangleright$	0	1	0	1	$\underline{\#}$	
s	$\triangleright$	0	1	0	1	0	$\underline{\#}$	$\triangleright$	0	1	0	1	0	$\underline{\#}$
q	$\triangleright$	0	1	0	1	$\underline{0}$		$\triangleright$	0	1	0	1	0	$\underline{\#}$
q	$\triangleright$	0	1	0	$\underline{1}$	0		$\triangleright$	0	1	0	1	0	$\underline{\#}$
q	$\triangleright$	0	1	$\underline{0}$	1	0		$\triangleright$	0	1	0	1	0	$\underline{\#}$
q	$\triangleright$	0	$\underline{1}$	0	1	0		$\triangleright$	0	1	0	1	0	$\underline{\#}$
q	$\triangleright$	$\underline{0}$	1	0	1	0		$\triangleright$	0	1	0	1	0	$\underline{\#}$
q	$\underline{\triangleright}$	0	1	0	1	0		$\triangleright$	0	1	0	1	0	$\underline{\#}$
p	$\triangleright$	$\underline{0}$	1	0	1	0		$\triangleright$	0	1	0	1	$\underline{0}$	$\#$
p	$\triangleright$	0	$\underline{1}$	0	1	0		$\triangleright$	0	1	0	$\underline{1}$	0	$\#$
p	$\triangleright$	0	1	$\underline{0}$	1	0		$\triangleright$	0	1	$\underline{0}$	1	0	$\#$
p	$\triangleright$	0	1	0	$\underline{1}$	0		$\triangleright$	0	$\underline{1}$	0	1	0	$\#$
p	$\triangleright$	0	1	0	1	$\underline{0}$		$\triangleright$	$\underline{0}$	1	0	1	0	$\#$
p	$\triangleright$	0	1	0	1	0	$\underline{\#}$	$\underline{\triangleright}$	0	1	0	1	0	$\#$
h_y	$\triangleright$	0	1	0	1	0	$\underline{\#}$	$\underline{\triangleright}$	0	1	0	1	0	$\#$

Definition: Sei $M = (Q, \Sigma, \delta, s)$ eine k-Band-Turingmaschine, $x \in (\Sigma - \{\#, \triangleright\})^*$ eine Eingabe für M und $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ ein Funktion.

- M berechnet $M(x)$ in t Schritten, falls

$\beta_0 = (s, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon,) \overset{M^t}{\longmapsto} (H, w_1, u_1, \ldots, w_k, u_k)$ mit $H \in \{h, h_y, h_n\}$.
(Falls $M(x) =$„↗", dann ist die Anzahl der Schritte unendlich.)

- M ist $f(n)$ *zeitbeschränkt*, falls für jede Eingabe $x \in (\Sigma - \{\#, \triangleright\})^*$ mit $|x| = n$, $M(x)$ in höchstens $f(n)$ Schritten berechnet wird.
- $f(n)$ ist eine *Zeitschranke* für M.

Definition: TIME($f(n)$) ist die Klasse aller Sprachen $L \subseteq (\Sigma - \{\triangleright, \#\})^*$, die mit einer Mehrband-Turingmaschine mit Zeitschranke $f(n)$ entschieden werden können.
Bemerkung:
TIME($f(n)$) ist eine *Komplexitätsklasse*.
Beispiel:
Sei L_p die Sprache alle Palindrome.
Unsere 1-Band-Turingmaschine zur Erkennung von Palindromen benötigte $(2n + 1) + (2n - 3) + (2n - 7) + \ldots$ Schritte um festzustellen, ob ein Wort der Länge n ein Palindrom ist oder nicht.
Somit ist $L_p \in \text{TIME}\left(\frac{(n+1)(n+2)}{2}\right)$.
Unsere 2-Band-Turingmaschine zur Erkennung von Palindromen benötigte $f'(n) = 3n + 4$ Schritte um festzustellen, ob ein Wort der Länge n ein Palindrom ist oder nicht.
Somit ist $L_p \in \text{TIME}(3n + 4)$.
Satz: Für jede k-Band-Turingmaschine $M = (Q, \Sigma, \delta, s)$ mit Zeitschranke $f(n) \geq n$ gibt es eine 1-Band-Turingmaschine $M' = (Q', \Sigma', \delta', s')$ mit Zeitschranke $g(n) \in O(f^2(n))$, so dass $\forall\ x \in (\Sigma - \{\#, \triangleright\})^* : M(x) = M'(x)$.

Beweis: Wir simulieren mit M' alle k Bänder von M auf einem Band.

Sei $\Sigma'' = \Sigma \cup \{\triangleleft, \triangleright', \triangleleft', \#'\}$, sei $\underline{\Sigma''} = \{\underline{a} \mid a \in \Sigma''\}$, und sei $\Sigma' = \Sigma'' \cup \underline{\Sigma''}$.

Eine Konfiguration

$$(q, w_1, u_1, \ldots, w_k, u_k)$$

für M entspricht einer Konfiguration

$$(q', \triangleright, w_1'u_1 \triangleleft' w_2'u_2 \triangleleft' \ldots w_k'u_k \triangleleft' \triangleleft)$$

für M', wobei w_i' das Wort w_i ist, in dem $\triangleright$ durch $\triangleright'$ und das letzte Symbol a_i durch $\underline{a_i}$ ersetzt wurde.
Das letzte Symbol $\triangleleft$ kennzeichnet das Bandende für M'.

1. M' ersetzt die Eingabe x durch $\triangleright' x \triangleleft' (\triangleright' \triangleleft')^{k-1} \triangleleft$.

   Dies kann mit zusätzlichen $O(k + |\Sigma|)$ Zuständen erreicht werden.

2. Zur Simulation eines Schrittes der k-Band-Turingmaschine M läuft M' von $\triangleright$ nach $\triangleleft$ und zurück und merkt sich dabei im aktuellen Zustand den Zustand von M und die Symbole der k Köpfe.

   Dies kann mit zusätzlichen $O(|Q| \cdot |\Sigma|^k)$ Zuständen erreicht werden.

3. M' kann nun von links nach rechts an den Kopfpositionen die entsprechenden (maximal zwei) Symbole ändern. Muß auf einem Band ein Leerzeichen eingeführt werden, so wird $\triangleleft'$ mit $\#'$ überschrieben, alles rechts von $\#'$ eine Position nach rechts verschoben, $\#'$ mit $\underline{\#}$ überschrieben und das Symbol rechts von $\#'$ mit $\triangleleft'$ überschrieben.

   Dies kann mit einer Zustandserweiterung um einen Faktor aus $O(k + |\Sigma|)$ erreicht werden.

4. Wenn M hält, ersetzt M' die Ausgabe $\triangleright' y_1 \triangleleft' \triangleright' y_2 \triangleleft' \cdots \triangleright' y_k \triangleleft' \triangleleft$ von M durch y_k und hält im Zustand h.

   Dies kann mit zusätzlichen $O(|\Sigma|)$ Zuständen erreicht werden.

Da M in Zeit $f(n)$ arbeitet, werden die k Wörter nicht länger als $f(|x|) + 1$ und somit ist das Wort auf dem Band von M' nicht länger als $k \cdot (f(|x|) + 2) + 2$.
M' läuft für jeden Schritt von M zweimal von links nach rechts und zurück und höchstens einmal zusätzlich nach rechts und zurück für jedes Leerzeichen, das eingefügt werden muß.
Daraus resultiert für jeden Simulationsschritt (für konstantes k) eine Laufzeit aus $O(f(|x|))$ und somit eine totale Laufzeit aus $O(f(|x|)^2)$. $\square$
Satz: Sei $L \in \text{TIME}(f(n))$. Für jedes $\epsilon > 0$ ist $L \in \text{TIME}(\epsilon \cdot f(n) + O(n))$.
Beweis: Sei $M = (Q, \Sigma, \delta, s)$ eine k-Band-Turingmaschine, die L in Zeit $f(n)$ entscheidet. Wir konstruieren eine k'-Band-Turingmaschine $M' = (Q', \Sigma', \delta', s')$, die L in Zeit $f'(n) = \epsilon \cdot f(n) + O(n)$ entscheidet. Für $k = 1$ sei $k' = 2$, ansonsten sei $k' = k$.
Idee: Wir kodieren mehrere Symbole aus Σ mit einem Symbol in Σ'. Dadurch kann M' in einer festen Anzahl von Schritten beliebig viele Schritte von M simulieren. Sei r die Anzahl der Symbole, die zusammengefaßt werden sollen, und $\Sigma' = \Sigma \cup (\Sigma \cup \underline{\Sigma})^r$.
Beispiel:
Sei $\Sigma = \{0, 1, \#, \triangleright\}$ und $r = 4$.
Eine Konfiguration

$$(q,\ \triangleright\ x_1\ x_2\ x_3,\ x_4\ x_5\ x_6\ x_7\ x_8\ x_9\ \#\ \ldots)$$

mit $x_i \in \Sigma$ für M entspricht einer Konfiguration

$$(q', \triangleright\ (x_1, x_2, \underline{x_3}, x_4),\ (x_5, x_6, x_7, x_8)\ (x_9, \#, \#, \#))$$

für M'.

M		M'	
q_1	$\triangleright\ x_1\ x_2\ \underline{x_3}\ x_4\ x_5\ x_6\ x_7\ x_8$	q_1'	$\triangleright\ \underline{(x_1, x_2, \underline{x_3}, x_4)}\ (x_5, x_6, x_7, x_8)$
q_2	$\triangleright\ x_1\ \underline{x_2}\ y_3\ x_4\ x_5\ x_6\ x_7\ x_8$		
q_3	$\triangleright\ x_1\ y_2\ \underline{y_3}\ x_4\ x_5\ x_6\ x_7\ x_8$		
q_4	$\triangleright\ x_1\ y_2\ z_3\ \underline{x_4}\ x_5\ x_6\ x_7\ x_8$		
q_5	$\triangleright\ x_1\ y_2\ z_3\ y_4\ \underline{x_5}\ x_6\ x_7\ x_8$	q_5'	$\triangleright\ (x_1, y_2, z_3, y_4)\ \underline{(x_5\ x_6, x_7, x_8)}$

M' schreibt zuerst die Eingabe x in umgewandelter Form auf das zweite Band. Eingabe 0110100101 wird zum Beispiel für $r = 4$ transformiert in

$$(0110)(1001)(01\#\#).$$

Das zweite Band wird jetzt als Eingabeband betrachtet.
M' simuliert in höchstens 6 Schritten r Schritte von M. Dazu bewegt M' alle Köpfe eine Position nach rechts, zwei Positionen nach links und wieder eine Position nach rechts.
Dabei merkt sich M' im Zustand alle Symbole links und rechts von den Köpfen und kann nun in 2 weiteren Schritten r Schritte von M simulieren. (r Schritte von M können maximal 2 Symbole pro Band für M' verändern.)
Wenn M hält, dann hält auch M'.
M' hält auf Eingabe x nach $O(|x|) + 6\lceil \frac{f(|x|)}{r} \rceil$ Schritten.
Mit $r \geq \lceil \frac{6}{\epsilon} \rceil$, folgt $L \in \text{TIME}(\epsilon \cdot f(n) + O(n))$. $\square$

Beispiele für platzbeschränkte Berechnungen:

Sei

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \overset{M^*}{\longmapsto} (H, w_1, u_1, \ldots)$$

mit $H \in \{h, h_y, h_n\}$ eine Berechnung.

Sei $\sum_{i=1}^{k} \mid w_i u_i \mid$ der für die Berechnung insgesamt benötigte Platz.

Dann können Palindrome x auf Platz $O(|x|)$ entschieden werden.

Die Definition einer Platzschranke ohne Berücksichtigung der Eingabe und Ausgabe kann jedoch zu besseren Ergebnissen führen.

Palindrome können dann zum Beispiel auf Platz $O(\log(n))$ erkannt werden.

Definition: Eine *k-Band-Turingmaschine mit Ein- und Ausgabe* ist eine k-Band-Turingmaschine $M = (Q, \Sigma, \delta, s)$, in der das Programm δ wie folgt eingeschränkt ist. Sei $\delta(q, a_1, \ldots, a_k) = (q', b_1, D_1, \ldots, b_k, D_k)$, dann gilt

- $a_1 = b_1$ (das erste Band wird nie verändert),
- Falls $a_1 = \#$, dann ist $D_1 \in \{\leftarrow, \perp\}$ (die Eingabe wird nicht verlassen) und
- $D_k \in \{\perp, \rightarrow\}$ (auf dem letzten Band kann links vom Kopf nicht korrigiert werden).

Definition: Sei $M = (Q, \Sigma, \delta, s)$ eine k-Band-Turingmaschine mit Ein- und Ausgabe, $x \in (\Sigma - \{\#, \triangleright\})^*$ eine Eingabe für M und $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ ein Funktion.

- M berechnet $M(x)$ auf Platz t, falls

  $\beta_0 = (s, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon,) \overset{M^*}{\longmapsto} (H, w_1, u_1, \ldots, w_k, u_k)$ mit $H \in \{h, h_y, h_n\}$ und $t = \sum_{i=2}^{k-1} (\mid w_i u_i \mid -1)$.
- M ist $f(n)$ *platzbeschränkt*, falls für jede Eingabe $x \in (\Sigma - \{\#, \triangleright\})^*$ mit $|x| = n$, $M(x)$ auf Platz $\leq f(n)$ berechnet wird.

Bemerkungen:
$f(n)$ ist eine *Platzschranke* für M.
SPACE($f(n)$) ist die Klasse aller Sprachen $L \subseteq (\Sigma - \{\#, \triangleright\})^*$, die mit einer $f(n)$ platzbeschränkten Turingmaschine M (mit Ein- und Ausgabe) entschieden werden können.

Satz: Sei $L \in \text{SPACE}(f(n))$. Für jedes $\epsilon > 0$ ist $L \in \text{SPACE}(\epsilon \cdot f(n) + O(1))$.

Beweis: Analog zum Speed-Up Theorem.

Definition:

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

$$\text{PSPACE} = \bigcup_{k \geq 1} \text{SPACE}(n^k)$$

20.2 Nichtdeterministische Turingmaschinen

Definition: Eine *nichtdeterministische Turingmaschine* ist eine Turingmaschine $M = (Q, \Sigma, \Delta, s)$ mit einem Programm Δ der Form

$$\Delta \subseteq (Q \times \Sigma) \times ((Q \cup \{h_y, h_n, h\}) \times \Sigma \times \{\leftarrow, \perp, \rightarrow\}).$$

- Ein Konfigurationspaar $(q, w, u), (q', w', u')$ heißt *Berechnungsschritt*, bezeichnet mit $(q, w, u) \overset{M}{\longmapsto} (q', w', u')$, falls für ein $w'' \in \Sigma^*$ und ein $a \in \Sigma$, $w = w''a$, und $((q, a), (q', b, D)) \in \Delta$ mit:
  - $D =$ '$\perp$' und $w' = w''b$ und $u' = u$, oder
  - $D =$ '$\leftarrow$' und $w' = w''$ und $u' = bu$, oder
  - $D =$ '$\rightarrow$' und
    $w' = w''b\#$ und $u' = \epsilon$ (falls $u = \epsilon$) oder
    $w' = w''bc$ und $u' = u''$ (falls $u = cu''$ für ein $c \in \Sigma$ und ein $u'' \in \Sigma^*$).
- Sei $\overset{M^k}{\longmapsto}$ eine Berechnung in k Schritten und $\overset{M^*}{\longmapsto}$ der transitive Abschluss von $\overset{M}{\longmapsto}$ (eine Berechnung in endlich vielen Schritten)
- M entscheidet eine Sprache $L \subseteq (\Sigma - \{\#, \triangleright\})^*$, falls $\forall x \in (\Sigma - \{\#, \triangleright\})^*$:

$$x \in L \quad \Leftrightarrow \quad (s, \triangleright, x) \overset{M^*}{\longmapsto} (h_y, w, u).$$

- Sei $f : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. M entscheidet eine Sprache L in Zeit $f(n)$, wenn M die Sprache L entscheidet und $\forall x \in (\Sigma - \{\#, \triangleright\})^*$:

  Falls $(s, \triangleright, x) \overset{M^k}{\longmapsto} (q, w, u)$ für ein $q \in Q \cup \{h_y, h_n\}$, dann ist $k \leq f(|x|)$.
- Die Klasse der Sprachen, die von nichtdeterministischen Turingmaschinen in Zeit $f(n)$ entschieden werden, bezeichnen wir mit

$$\mathrm{NTIME}(f(n)).$$

Definition:

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

Offensichtlich gilt : $\mathrm{P} \subseteq \mathrm{NP}$.
Satz: Sei $f : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion.

$$\mathrm{NTIME}(f(n)) \subseteq \bigcup_{c > 1} \mathrm{TIME}(c^{f(n)}).$$

Beweis: Sei $M = (Q, \Sigma, \Delta, s)$ eine nichtdeterministische Turingmaschine mit Zeitschranke $f(n)$.
Sei $C_{(q,a)} = \{(q', b, D) \mid ((q, a), (q', b, D)) \in \Delta\}$.
Sei

$$\alpha = \max_{q \in Q, a \in \Sigma} \mid C_{(q,a)} \mid .$$

M kann wie folgt mit einer 3-Band-Turingmaschine M' simuliert werden:

- M' zählt auf Band 3 einen α-nären Zähler mit den Ziffern $c_1 c_2 c_3 \cdots$ fortlaufend hoch.

  Nach jeder Zählererhöhung wird die Eingabe x von Band 1 auf Band 2 kopiert und die Maschine M auf Eingabe x wie folgt simuliert:

  > Wenn M im l-ten Schritt im Zustand q das Symbol a liest, dann wird in der Simulation von M das c_l-te Tripel aus $C_{(q,a)}$ gewählt.

- M' hält im Zustand h_y bzw. h_n, wenn bei der Simulation von M der Zustand h_y bzw. h_n erreicht wird.

$\alpha^{f(n)}$ Simulationen mit höchstens $f(n)$ Schritten ergibt eine Simulationszeit aus $O(c^{f(n)})$ für $c = \alpha$. $\square$
Definition: Eine nichtdeterministische k-Band-Turingmaschine $M = (Q, \Sigma, \Delta, s)$ mit *Ein- und Ausgabe* entscheidet eine Sprache L auf Platz $f(n)$, wenn M die Sprache L entscheidet und $\forall\ x \in (\Sigma - \{\triangleright, \#\})^*$ mit $(s, \triangleright, x, \ldots, \triangleright, \epsilon) \overset{M^*}{\longmapsto} (q, w_1, u_1, \ldots, w_k, u_k)$ gilt:

$$\sum_{i=2}^{k-1} (\mid w_i u_i \mid -1) \leq f(\mid x \mid).$$

Die Klasse der Sprachen, die von nichtdeterministischen Turingmaschinen auf Platz $f(n)$ entschieden werden, bezeichnen wir mit

$$\text{NSPACE}(f(n)).$$

Definition:

$$\text{NPSPACE} = \bigcup_{k \geq 1} \text{NSPACE}(n^k)$$

Offensichtlich gilt: PSPACE $\subseteq$ NPSPACE. (Es gilt sogar PSPACE = NPSPACE.)

20.3 NP-Vollständigkeit

Bemerkungen:
Für jede Sprache $L \in \text{NP}$ gibt es ein Polynom p, so dass $L \in \text{TIME}(2^{p(n)})$.
Sei $\text{P} \neq \text{NP}$, dann gilt:

1. Alle Probleme in $\text{NP} - \text{P}$ sind inhärent schwer.
2. Alle Probleme in P sind effizient lösbar.

Definition: Eine *polynomielle Transformation* von einer Sprache $L_1 \subseteq \Sigma_1^*$ auf eine Sprache $L_2 \subseteq \Sigma_2^*$ ist eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$ mit:

- f ist mit einer (deterministischen) Turingmaschine in polynomieller Zeit berechenbar.
- $\forall\ x \in \Sigma_1^* :\ x \in L_1 \Leftrightarrow f(x) \in L_2$.

L_1 ist in polynomieller Zeit auf L_2 reduzierbar, geschrieben $L_1 \leq_p L_2$, wenn es eine polynomielle Transformation von L_1 nach L_2 gibt.
Lemma:

$$L_1 \leq_p L_2,\ L_2 \in P\ \Rightarrow\ L_1 \in P$$

Beweis: Sei M_f die Turingmaschine, die eine polynomielle Transformation $f : \Sigma_1^* \to \Sigma_2^*$ berechnet.
Sei M_2 die Turingmaschine, die L_2 entscheidet.
Konstruiere eine Maschine M_1, die L_1 entscheidet, indem zuerst die Instanz x für M_1 in $f(x)$ für M_2 umgewandelt und anschließend $M_2(f(x)) = M_1(x)$ berechnet wird.
M_1 hat eine polynomielle Laufzeit, wenn M_f und M_2 polynomielle Laufzeiten haben, da $|f(x)|$ polynomiell in $|x|$ ist. $\square$
Lemma:

$$L_1 \leq_p L_2 \text{ und } L_2 \leq_p L_3\ \Rightarrow\ L_1 \leq_p L_3$$

Beweis: Nach Voraussetzung gibt es zwei Transformationen $f_1 : \Sigma_1^* \to \Sigma_2^*$ und $f_2 : \Sigma_2^* \to \Sigma_3^*$, die in polynomieller Zeit berechenbar sind. Seien p_1 bzw. p_2 die entsprechenden Polynome.

Nun gilt: $x \in L_1 \Leftrightarrow f_1(x) \in L_2$ und $x \in L_2 \Leftrightarrow f_2(x) \in L_3$ und somit $x \in L_1 \Leftrightarrow f_2(f_1(x)) \in L_3$.

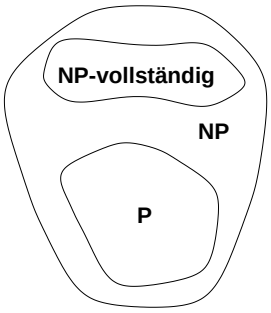
Da $|f_1(x)| \leq p_1(|x|)$ und $|f_2(x)| \leq p_2(|x|)$, gilt

$$|f_2(f_1(x))| \leq p_2(p_1(|x|)).$$

Also ist $f_2 \circ f_1$ eine polynomielle Transformation. $\square$
Definition: Eine Sprache L heißt *NP-schwer*, wenn $\forall L' \in \text{NP} : L' \leq_p L$.
Eine NP-schwere Sprache L heißt *NP-vollständig*, wenn sie aus NP ist.
Satz: Sei L eine NP-vollständige Sprache:

1. Falls $L \in \text{P}$, dann ist $\text{P} = \text{NP}$.
2. Falls $L \notin \text{P}$, dann sind alle NP-vollständigen Sprachen nicht in P.

Beweis: Sei $L \in \text{P}$. Da L NP-vollständig ist, sind alle Sprachen $L' \in \text{NP}$ aufgrund der Transitivität der polynomiellen Reduktion in P.
Sei $L \notin \text{P}$. Angenommen es gibt ein NP-vollständiges Problem L' aus P. Dann gibt es eine polynomielle Reduktion von $L \in \text{NP}$ nach L'. Da $L' \in \text{P}$, ist auch $L \in \text{P}$. (Widerspruch) $\square$



Da bis heute nicht bekannt ist, ob P = NP gilt, sind für uns alle NP-schweren Probleme (Sprachen) nicht effizient entscheidbar.

20.4 Satisfiability ist NP-vollständig

Definition: Sei $X = \{x_1, \ldots, x_n\}$ eine Menge *Boolescher Variablen*.
Eine *Belegung* für X ist eine Funktion $t : X \to \{T, F\}$ mit

$$t(x) = \begin{cases} F & \text{falls } x \text{ falsch ist} \\ T & \text{falls } x \text{ wahr ist} \end{cases}$$

Für eine Variable $x \in X$ seien x und $\overline{x}$ *Literale* über X.
Literal x ist wahr, falls $t(x) = T$, Literal $\overline{x}$ ist wahr, falls $t(x) = F$.
Eine *Klausel* C ist eine Menge von Literalen.
Klausel C wird von einer Belegung t für X *erfüllt*, wenn C mindestens ein Literal enthält, das unter t wahr ist.
Eine Menge F von Klauseln über X ist erfüllbar, wenn es eine Belegung t für X gibt, die jede Klausel erfüllt.
Problem: SATISFIABILITY
Gegeben: Eine Menge X von Variablen und eine Menge F von Klauseln über X.

Frage: Ist F erfüllbar, d.h. gibt es eine Belegung t der Variablen in X, die alle Klauseln erfüllt?
Beispiel:
$X = \{x_1, x_2, x_3\}$,
$C_1 = \{x_1, \overline{x_2}, \overline{x_3}\},\ C_2 = \{\overline{x_1}, \overline{x_2}, x_3\},\ C_3 = \{x_1, \overline{x_2}, x_3\}$,
$F = \{C_1, C_2, C_3\}$
Folgende Belegung $t : X \to \{T, F\}$ für X erfüllt alle Klauseln in F:

$$t(x_1) = T, \quad t(x_2) = F, \quad t(x_3) = T$$

Satz: [Cook's Theorem] SATISFIABILITY ist NP-vollständig.
(Die Menge SAT aller erfüllbaren SATISFIABILITY Instanzen ist eine NP-vollständige Menge.)
Beweis: Sei X eine Menge von Variablen, F eine Menge von Klauseln über X und

$$\text{SAT} = \left\{ (X, F) \;\middle|\; \begin{array}{l} \text{es gibt eine Belegung } t : X \to \{T, F\} \text{ für } X, \\ \text{die alle Klauseln in } F \text{ erfüllt} \end{array} \right\}$$

Wir zeigen, dass SAT eine NP-vollständige Sprache ist.

1. SAT $\in$ NP:

   Eine nichtdeterministische Turingmaschine M kann für jede Variable aus X einen Belegungswert aus $\{T, F\}$ „raten“ und anschließend mit der ausgewählten Belegung alle Klauseln überprüfen.

   Sei β_0 die Startkonfiguration, $t : X \to \{T, F\}$ eine Belegung für X und β_t die Konfiguration, die Belegung t repräsentiert.

   Wichtig:

   (a) Für jede mögliche Belegung t muss eine Berechnungen $\beta_0 \to \beta_t$ existieren (Auswahlphase).

   (b) Es muss genau dann eine Berechnung von β_t zu einer akzeptierenden Haltekonfiguration geben, wenn Belegung t alle Klauseln erfüllt (Verifikationsphase).

   (c) Die Längen der akzeptierenden und ablehnenden Berechnungen müssen alle polynomiell in der Größe der Eingabe (= Anzahl der Variable + Anzahl der Literale in allen Klauseln) sein.

2. SAT ist NP-schwer:

   Sei $L \in$ NP eine beliebige Sprache aus NP und $M = (Q, \Sigma, \Delta, s)$ eine nichtdeterministische Turingmaschine, die L in Zeit $p(n)$ entscheidet, für ein Polynom $p : \mathbb{N}_0 \to \mathbb{N}_0$.

Wir konstruieren in polynomieller Zeit für eine beliebige Eingabe $x \in (\Sigma - \{\triangleright, \#\})^*$ eine Instanz (X, F) für SATISFIABILITY, die genau dann erfüllbar ist „$(X, F) \in SAT$“, wenn $x \in L$.

Dies entspricht einer polynomiellen Transformation von L nach SAT!

Vorüberlegung:

Wir betrachten einen *Berechnungrahmen* von M auf Eingabe x mit $n = |x|$ als ein Wort der Form $\#'\beta_0\#'\beta_1\#' \ldots \#'\beta_{p(n)-1} \in \overline{\Sigma}^*$ mit folgenden Eigenschaften:

(a) Jedes β_i beschreibt eine Konfiguration von M und besteht aus genau $p(n)$ vielen Symbolen, dass heisst $\#'\beta_0\#'\beta_1\#' \ldots \#'\beta_{p(n)-1}$ hat die Länge $(p(n)+1) \cdot p(n)$.

(b) Sei $\overline{\Sigma} := \Sigma \cup \{\#'\} \cup \Delta$.

(c) Das Symbol #' wird nur zur Trennung der Konfigurationen $\beta_0, \ldots, \beta_{p(n)-1}$. benötigt

(d) In jeder Konfiguration β_i gibt es genau ein Symbol $(q, a, q', a', D) \in \Delta$. welches die Kopfposition von M in β_i markiert; dabei ist q der aktuelle Zustand, a das Symbol auf der Kopfposition und (q, a, q', a', D) der in dieser Konfiguration gewählte Programmschritt.

Falls M die Eingabe x in einer betrachteten Berechnung in weniger als $p(n)$ vielen Schritten entscheidet, wird die letzte Konfiguration (die Konfiguration, die im nächsten Schritt zu einer Haltekonfiguration führen würde) einfach wiederholt.

Erzeuge für jede Position i, $1 \leq i \leq (p(n)+1) \cdot p(n)$, und jedes Symbol $s \in \overline{\Sigma}$ eine Boolesche Variable $c_{i,s}$ für X, die genau dann wahr sein soll, wenn in einer betrachteten Berechnung das Symbol s auf Position i steht.

Die Klauselmenge F wird nun genau so konstruiert, dass die wahren Variablen $c_{i,s}$ in X eine akzeptierende Berechnung von M auf Eingabe x darstellen.

(a) Für jede Position i soll genau ein $c_{i,s}$ wahr sein (weil auf einer Position nur ein Symbol stehen soll).
Sei $\overline{\Sigma} = \{s_1, \ldots, s_l\}$.
Dann erhält F für jedes i, $1 \leq i \leq (p(n)+1) \cdot p(n)$, eine Klausel

$$\{c_{i,s_1}, c_{i,s_2}, \ldots, c_{i,s_l}\}$$

und für alle $a, b \in \overline{\Sigma}$ mit $a \neq b$ eine Klausel

$$\{\overline{c_{i,a}}, \overline{c_{i,b}}\}.$$

(b) Sei $x = x_1 x_2 \dots x_n \in \Sigma$ die Eingabe für M.
β_0 soll die initiale Konfiguration sein.
Deshalb erhält F_x die Klauseln:
$\{c_{1,\#'}\}, \{c_{2,(q,a,q',a',D)} \mid (q,a,q',a',D) \in \Delta \text{ mit } q = s, a = \triangleright\}$,
$\{c_{3,x_1}\}, \{c_{4,x_2}\}, \dots, \{c_{n+2,x_n}\}$,
$\{c_{n+3,\#}\}, \{c_{n+4,\#}\}, \dots, \{c_{p(n)+1,\#}\}$.

(c) Die letzte Konfiguration soll akzeptierend sein.
Deshalb hat F_x die Klausel:

$$\left\{ c_{i,(q,a,h_y,a',D)} \;\middle|\; \begin{array}{l} p(n) \cdot p(n) < i \leq (p(n)+1) \cdot p(n), \\ (q,a,q',a',D) \in \Delta \text{ mit } q' = h_y \end{array} \right\}.$$

(d) Wir betrachten nun die Funktion $f_M : \overline{\Sigma}^3 \to \overline{\Sigma}$ mit $f(a,b,c)$ ist das Symbol an einer Position j in Konfiguration β_{i+1}, wenn a, b, c die Symbole an den Positionen $j-1, j$ und $j+1$ in der direkten Vorgängerkonfiguration β_i sind.
Zusätzlich gilt $f_M(a,b,c) = b$ falls $b = \#'$ oder falls a, b oder c ein Symbol aus Δ ist, welches einen Endzustand enthält.
Konfiguration β_{i+1} soll eine direkte Nachfolgekonfiguration von β_i sein.
Dies kann nun mit Hilfe von f_M wie folgt als Boolscher Ausdruck definiert werden:
$\forall j : (i \cdot (p(n)+1) < j < ((i+1) \cdot (p(n)+1) : \forall a,b,c \in \overline{\Sigma}$

$$\bigvee [c_{j-1,a} \wedge c_{j,b} \wedge c_{j+1,b} \wedge c_{j+(p(n)+1),f_M(a,b,c)}]$$

Durch Ausmultiplizieren entstehen für jedes j höchstens $4^{4^{|\Delta|}}$ Klauseln mit je $4^{|\Delta|}$ Literalen.

Die Variablen $c_{i,s}$ definieren nun genau dann eine akzeptierende Berechnung von M auf Eingabe x, wenn $x \in L$.

(X, F) kann offensichtlich in polynomieller Zeit konstruiert werden. □

Lemma: Sei $L_2 \in$ NP und $L_1 \leq_p L_2$ für ein NP-vollständiges Problem L_1. Dann ist L_2 NP-vollständig.
Beweis: Wenn L_1 NP-vollständig ist, dann gilt $\forall L' \in \text{NP} : \; L' \leq_p L_1$.
Aus der Transitivität der polynomiellen Reduktion $\leq_p$ und aus $L_1 \leq_p L_2$ folgt:
$\forall\, L' \in \text{NP} : L' \leq_p L_1 \leq_p L_2 \;\Rightarrow\; L' \leq_p L_2$. □
Problem: SAT
Gegeben: Eine Menge X von Variablen und eine Menge $F = \{C_1, \dots, C_m\}$ von

Klauseln über X, wobei jede Klausel genau 3 Literale enthält.
Frage: Gibt es eine Belegung t der Variablen, die alle Klauseln in F erüllt?
Satz: 3SAT ist NP-vollständig.
Beweis:

1. 3SAT $\in$ NP: Siehe SAT $\in$ NP.

2. 3SAT ist NP-schwer:

   Wir zeigen: SAT $\leq_p$ 3SAT

   Vorgehensweise: Wir geben einen polynomiellen Algorithmus an, der aus einer beliebigen Instanz (X, F) für SAT eine Instanz (X', F') für 3SAT erzeugt, so dass F genau dann erfüllbar ist, wenn F' erfüllbar ist.

   Sei $F = \{C_1, \ldots, C_m\}$.

   Für jede Klausel $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,k}\}$ $(1 \leq i \leq m)$ konstruieren wir mit zusätzlichen Hilfsvariablen Klauseln $C'_{i,1}, C'_{i,2}, \ldots, C'_{i,l}$ für F', so dass C_i genau dann erfüllbar ist, wenn alle $C'_{i,j}$ erfüllbar sind.

   Sei $C_i = \{c_{i,1}, \ldots, c_{i,k}\}$, dann erzeugen wir folgende Klauseln für F'.

k	Klauseln für F'
1	$\{c_{i,1}, y_{i,1}, y_{i,2}\}, \{c_{i,1}, \overline{y_{i,1}}, y_{i,2}\}, \{c_{i,1}, y_{i,1}, \overline{y_{i,2}}\}, \{c_{i,1}, \overline{y_{i,1}}, \overline{y_{i,2}}\}$
2	$\{c_{i,1}, c_{i,2}, y_{i,1}\}, \{c_{i,1}, c_{i,2}, \overline{y_{i,1}}\}$
3	$\{c_{i,1}, c_{i,2}, c_{i,3}\}$
> 3	$\{c_{i,1}, c_{i,2}, y_{i,1}\}, \{c_{i,3}, \overline{y_{i,1}}, y_{i,2}\}, \{c_{i,4}, \overline{y_{i,2}}, y_{i,3}\}, \ldots$
	$\{c_{i,k-2}, \overline{y_{i,k-4}}, y_{i,k-3}\}, \{c_{i,k-1}, c_{i,k}, \overline{y_{i,k-3}}\}$

   Hierbei sind $X_i = \{y_{i,1}, y_{i,2}, \ldots, y_{i,k-3}\}$ neue Variablen.

   Die Variablen für F' sind $X' = X \cup X_1 \cup X_2 \cup \ldots \cup X_m$.

   Diese Transformation ist offensichtlich in polynomieller Zeit möglich. $\square$

Bemerkung:
Ein *Entscheidungsproblem* ist ein Problem, für das jede Eingabe entweder mit „ja" oder mit „nein" beantwortet wird.
Ein Entscheidungsproblem kann als eine Sprache L aufgefaßt werden, welche durch die Menge aller ja-Eingaben definiert ist. Somit läßt sich der Begriff der NP-Vollständigkeit auf Entscheidungsprobleme übertragen.

20.5 NP-vollständige Graphenprobleme

Definition: Sei $G = (V, E)$ ein ungerichteter Graph und $V' \subseteq V$ eine Knotenmenge.

1. V' ist eine *vollständige Menge*, falls für alle $u, v \in V'$, $\{u, v\} \in E$.

2. V' ist eine *unabhängige Menge*, falls für alle $u, v \in V'$, $\{u, v\} \notin E$.

3. V' ist eine *Knotenüberdeckung*, falls für alle $\{u, v\} \in E$, $u \in V'$ oder $v \in V'$.

Problem: CLIQUE
Gegeben: Ungerichteter Graph $G = (V, E)$ und eine Zahl k, $1 \leq k \leq |V|$.
Frage: Gibt es in G eine vollständige Menge $V' \subseteq V$ der Größe $\geq k$.
Satz: CLIQUE ist NP-vollständig.
Beweis:

1. CLIQUE $\in NP$

   Idee: Wähle k Knoten und überprüfe, ob alle Knoten paarweise mit einer Kante verbunden sind.

   Das geht mit einer nichtdeterministischen Turingmaschinen $M = (Q, \Sigma, \Delta, s)$ in polynomieller Zeit!

   Das Programm Δ wird so entworfen, dass es jeden Knoten wählen bzw. nicht wählen kann und die ausgewählten Knoten auf das zweite Band schreibt (Auswahlphase). Die Überprüfung, ob mindesten k Knoten gewählt wurden und ob sie alle miteinander verbunden sind, erfolgt anschließend deterministisch (Verifikationsphase).

   (a) Wenn die Maschine M mit Eingabe G, k die Eingabe akzeptiert ($M(< G, k >) =$„Yes“), dann gibt es eine eine Clique der Größe k.

   (b) Wenn es eine Clique der Größe k gibt, dann besitzt die Maschine eine akzeptierende Berechnung.

   (c) Die nichtdeterministische Maschine hat eine polynomielle Laufzeit.

   $\Longrightarrow$ M entscheidet in polynomieller Zeit die Sprache

   $$\left\{ (G,k) \;\middle|\; \begin{array}{l} G = (V,E) \text{ ist ein ungerichteter Graph }, k \in \{1, \ldots, |V|\}, \\ \text{es gibt eine vollständige menge } V' \subseteq V \text{ der Größe k} \end{array} \right\}$$

2. CLIQUE ist NP-schwer

   Wir zeigen: 3SAT $\leq_p$ CLIQUE

   Sei $X = \{x_1, \ldots, x_n\}, F = \{C_1, \ldots, C_m\}$ mit $|C_i| = 3$ eine Instanz für 3SAT.

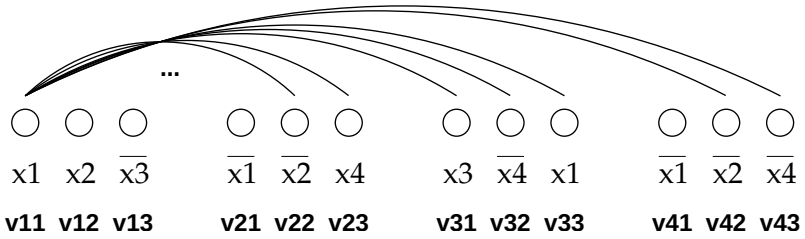
   Gesucht ist eine polynomielle Reduktion, die aus (X, F) einen Graphen G und eine Zahl m konstruiert, so dass G genau dann eine m-Clique hat, wenn F erfüllbar ist.

   Sei $C_i = \{c_{i,1}, c_{i,2}, c_{i,3}\}$.

Sei $G = (V, E)$ mit
$V = \{v_{i,j} \ : \ 1 \leq i \leq m, \ 1 \leq j \leq 3\}$ und
$E = \{\{v_{i,j}, v_{i',j}\} \mid i \neq i'$ und die entsprechenden Literale $c_{i,j}, c_{i',j'}$ sind nicht komplementär. $\}$

Beispiel:

$X = \{x_1, x_2, x_3, x_4\}$, $F = \{x_1, x_2, \overline{x_3}\}, \{\overline{x_1}, \overline{x_2}, x_4\}, \{x_3, \overline{x_4}, x_1\}, \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}$



Der Graph G kann offensichtlich in polynomieller Zeit aus F konstruiert werden. G hat nun genau dann eine vollständige Menge der Größe m, wenn F erfüllbar ist. $\square$

Problem: INDEPENDENT SET
Gegeben: Ungerichteter Graph $G = (V, E)$ und eine Zahl k, $1 \leq k \leq |V|$.
Frage: Gibt es in G eine unabhängige Menge $V' \subseteq V$ der Größe $\geq k$.
Satz: INDEPENDENT SET ist NP-vollständig.
Beweis:
$V \subseteq V'$ ist in G genau dann eine vollständige Menge, wenn V' in $\overline{G} = (V, \overline{E})$ eine unabhängige Menge ist, wobei $\overline{E} = (\{u, v\} \mid u, v \in V, \ u \neq v, \ \{u, v\} \notin E\}$. $\square$
Problem: VERTEX COVER
Gegeben: Ungerichteter Graph $G = (V, E)$ und eine Zahl k, $1 \leq k \leq |V|$.
Frage: Gibt es in G eine Knotenüberdeckung $V' \subseteq V$ der Größe $\leq k$.
Satz: VERTEX COVER ist NP-vollständig.
Beweis: $V' \subseteq V$ ist genau dann eine Knotenüberdeckung, wenn $V - V'$ eine unabhängige Menge ist. $\square$
Problem: Directed Hamiltonian Circuit (DHC)
Gegeben: Gerichteter Graph $G = (V, E)$.
Frage: Gibt es einen Kreis in G, der jeden Knoten genau einmal besucht?
Satz: DHC ist NP-vollständig
Beweis:

1. DHC $\in$ NP ist offensichtlich (rate einen Kreis und verifiziere).

2. DHC ist NP-schwer.

   Wir zeigen: 3SAT $\leq_p$ DHC

Sei $X = \{x_1, \ldots, x_n\}$ und $F = \{C_1, \ldots, C_m\}$ mit $|C_i| = 3$, $1 \leq i \leq m$, eine Instanz für 3-SAT.

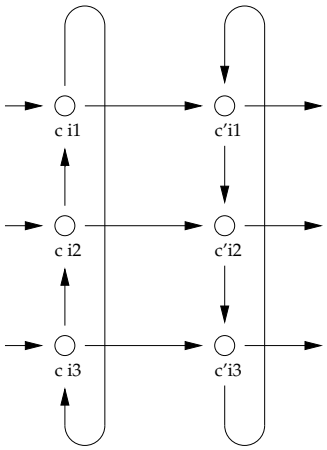
Gesucht ist eine polynomielle Reduktion, die aus (X, F) einen Graphen G konstruiert, der genau dann einen gerichteten Hamiltonkreis enthält, wenn F erfüllbar ist.

Sei $C_i = \{c_{i,1}, c_{i,2}, c_{i,3}\}$.

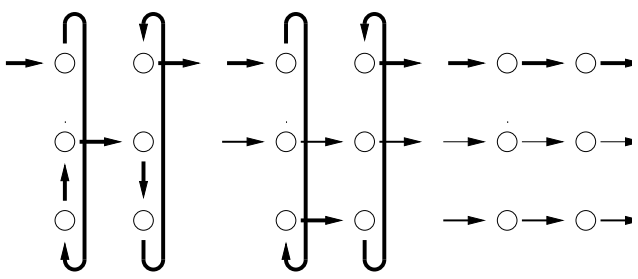
G erhält einen Knoten u_i für jede Variable $x_i \in X$ und eine Klauselkomponente mit 6 Knoten für jede Klausel $C_i \in F$.

Jede Knoten u_i bekommt genau zwei auslaufende Kanten, eine für Literal x_i und eine für Literal $\overline{x_i}$.

Aufbau der Klauselkomponente für $C_i = \{c_{i,1}, c_{i,2}, c_{i,3}\}$.



Die Klauselkomponenten sind so entworfen, dass alle 6 Knoten in einem, in zwei oder in drei Durchläufen besucht werden können. Wenn man über Knoten $c_{i,1}$ (bzw. $c_{i,2}, c_{i,3}$) in die Komponente hineingeläuft, so muß die Komponente über Knoten $c'_{i,1}$ (bzw. $c'_{i,2}, c'_{i,3}$) verlassen werden, ansonsten erhält man keinen Hamiltonkreis.

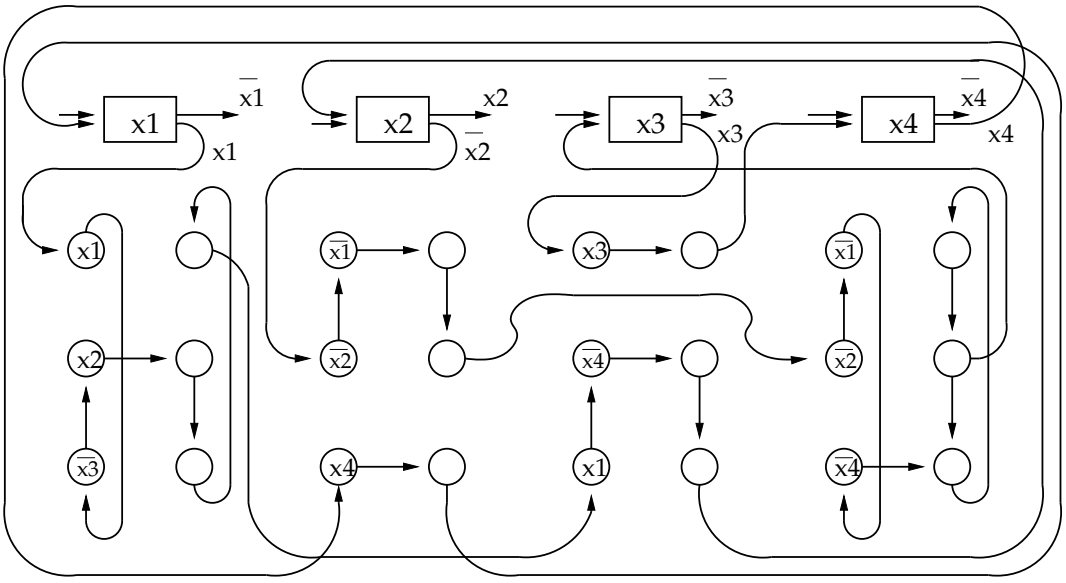


G bekommt die folgende zusätzlichen Kanten:

- Vom Knoten u_i zur ersten Klauselkomponente mit Literalknoten x_i;

- Vom Literalknoten x_i' zur nächsten Klauselkomponente mit Literalknoten x_i usw.
- Von der letzten Klauselkomponente mit Literalknoten x_i' zum Knoten x_{i+1} bzw. x_1 falls $i = n$.
  Das gleiche auch für die Literalknoten $\overline{x_i}$.

Beispiel:
$F = \{\{x_1, x_2, \overline{x_3}\}, \{\overline{x_1}, \overline{x_2}, x_4\}, \{x_3, \overline{x_4}, x_1\}, \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}\}$



Belegung: $x_1, \overline{x_2}, x_3, x_4$

Diese Transformation arbeitet offensichtlich in polynomieller Zeit. □

Problem: Directed Hamiltonian Path (DHP)
Gegeben: Gerichteter Graph $G = (V, E)$ und zwei Knoten $u, v \in V$.
Frage: Gibt es einen Weg in G von u nach v, der jeden Knoten genau einmal besucht?
Satz: DHP ist NP-vollständig
Beweis: Wie im Beweis von "DHC ist NP-vollständig'. Füge einen neuen Knotem u_{n+1} hinzu und leite die beiden Kanten, welche zum Knoten u_1 führen, um zum Knoten u_{n+1}. Nun gibt es genau dann einen Hamilton Weg von u_1 nach u_{n+1}, wenn die Instanz X, F für 3-SAT eine erfüllende Belegung hat. □
Problem: Hamilton Circuit (HC)
Gegeben: Ungerichteter Graph $G = (V, E)$.
Frage: Gibt es einen Kreis in G, der jeden Knoten genau einmal besucht?
Problem: Hamiltonian Path (HP)
Gegeben: Ungerichteter Graph $G = (V, E)$ und zwei Knoten $u, v \in V$.

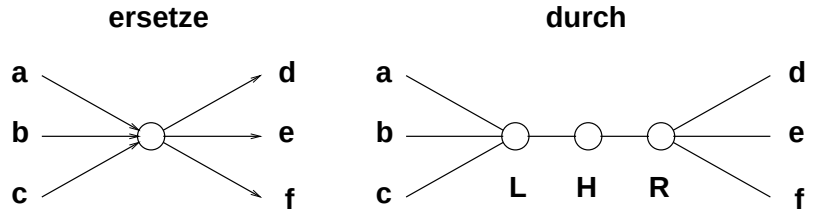
Frage: Gibt es einen Weg in G von u nach v, der jeden Knoten genau einmal besucht?
Satz: HC und HP sind NP-vollständig
Beweis:

1. HC $\in$ NP und HP $\in$ NP: Rate und verifiziere.

2. HC und HP sind NP-schwer:

   Wir zeigen: DHC $\leq_p$ HC bzw. DHP $\leq_p$ HP (durch lokale Ersetzung)

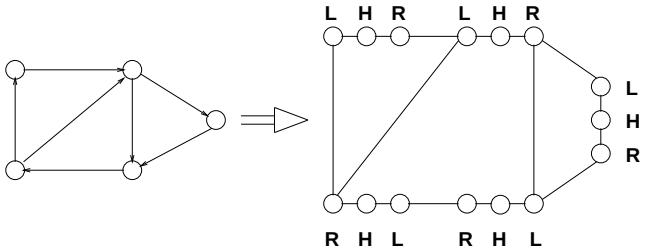
   Jeder knoten u wird durch drei Knoten u_L, u_H, u_R ersetzt. Knoten u_H wird mit u_L und u_R verbunden. Jede gerichtete Kante (u, v) wird durch eine ungereichtete Kante $\{u_R, v_L\}$ ersetzt.



   Der gerichtete Graph hat genau dann einen Hamiltonkreis, wenn der ungerichtete Graph einen Hamilton-Kreis hat, wobei jeder dritte Knoten ein Hilfsknoten ist.

   Der Gereichtete Graph hat genau dann einen Hamiltonweg von u_L nach v_R, wenn der ungerichtete Graph einen Hamiltonweg von u nach v hat. $\square$

Beispiel:



Problem: 3-DIMENSIONAL MATCHING (3-DM)
Gegeben: Drei disjunkte Mengen X, Y, Z mit je n Elementen und eine Relation

$$W \subseteq X \times Y \times Z$$

Frage: Gibt es eine Teilmenge $M \subseteq W$ mit $|M| = n$ (ein *Matching*), so dass jedes $x \in X$, jedes $y \in Y$ und jedes $z \in Z$ in genau einem Tripel aus M enthalten ist?
Satz: 3-DM ist NP-vollständig
Beweis:

1. 3-DM $\in$ NP: Rate M und verifiziere.

2. 3-DM ist NP-schwer:

   Wir zeigen: SAT $\leq_p$ 3-DM

   Sei $X = \{x_1, \ldots, x_n\}$ und $F = \{C_1, \ldots, C_m\}$, $1 \leq i \leq m$, eine Instanz für SAT.

   Gesucht ist eine polynomielle Reduktion, die aus (X, F) drei Mengen X, Y, Z mit gleicher Anzahl von Elementen und eine Relation $W \subseteq X \times Y \times Z$ konstruiert, so dass es genau dann ein Matching $M \subseteq W$ der Größe $|M| = |X| = |Y| = |Z|$ gibt, wenn F erfüllbar ist.

   Sei

$$\begin{aligned} X = & \quad \{x_{ij} \mid 1 \leq i \leq n,\ 1 \leq j \leq m\} \\ & \cup \ \{\overline{x}_{ij} \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}, \end{aligned}$$

$$\begin{aligned} Y = & \quad \{y_{ij} \mid 1 \leq i \leq n,\ 1 \leq j \leq m\} \\ & \cup \ \{s_j \mid 1 \leq j \leq m\} \\ & \cup \ \{a_k \mid 1 \leq k \leq m \cdot (n-1)\} \end{aligned}$$

   und

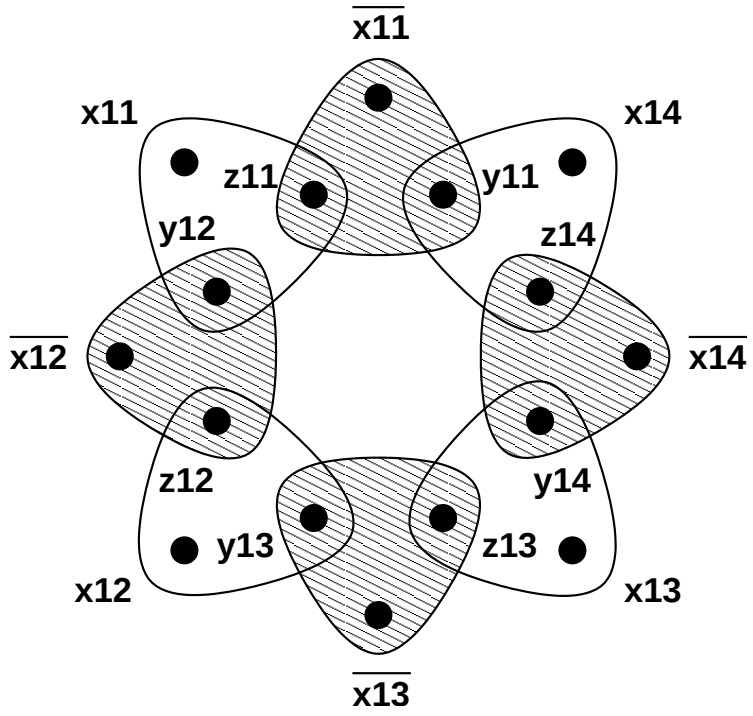
$$\begin{aligned} Z = & \quad \{z_{ij} \mid 1 \leq i \leq n,\ 1 \leq j \leq m\} \\ & \cup \ \{t_j \mid 1 \leq j \leq m\} \\ & \cup \ \{b_k \mid 1 \leq k \leq m \cdot (n-1)\}. \end{aligned}$$

   Für eine Variable x_i sei

$$T_i^1 = \{(\overline{x}_{ij}, y_{ij}, z_{ij}) \mid 1 \leq j \leq m\}$$

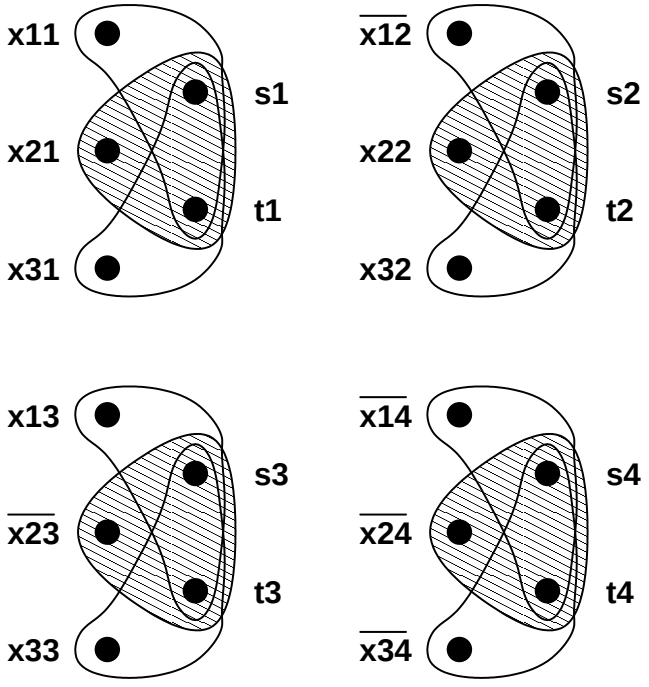
   und

$$T_i^0 = \{(x_{ij}, y_{ij+1}, z_{ij}) \mid 1 \leq j < m\} \cup \{x_{ij}, y_{i1}, z_{im}\}.$$



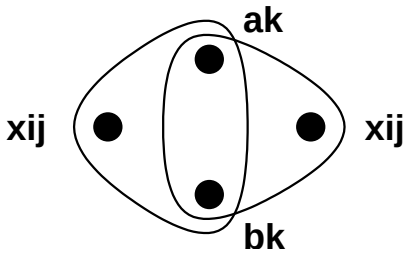
Für eine Klausel C_j sei:

$$R_j = \{(x_{ij}, s_j, t_j) \mid x_i \in C_j\} \cup \{(\overline{x_i}_j, s_j, t_j) \mid \overline{x_i} \in C_j\}.$$



Weiterhin sei

$$\begin{aligned} G = \quad & \{(x_{ij}, a_k, b_k) \mid 1 \le i \le n, 1 \le j \le m, 1 \le k \le m \cdot (n-1)\} \\ \cup \quad & \{(\overline{x_i}_j, a_k, b_k) \mid 1 \le i \le n, 1 \le j \le m, 1 \le k \le m \cdot (n-1)\} \end{aligned}$$



Die Tripelmenge W ist definiet durch

$$T_1^1 \cup \cdots \cup T_n^1 \cup T_1^0 \cup \cdots \cup T_n^0 \cup R_1 \cup \cdots \ldots \cup R_m \cup G.$$

Die Elemente y_{ij} und $z_{i,j}$ kommen nur in den Mengen T_i^1 und T_i^0 vor. Somit enthält jedes Matching $M \subseteq W$ entweder alle Tripel aus T_j^1 oder alle Tripel aus T_j^0. Diese Tripel fixieren die Belegung der Variablen.

Jeses Matching $M \subseteq W$ enthält für jede Klausel C_j genau ein Tripel aus R_j. Diese Tripel belegen die Existenz eines wahren Literals in jeder Klausel.

Die übrigen $m \cdot (n-1)$ Elemente $x_{ij}, \overline{x_{ij}}$ können mit den Tripeln aus G aufgefangen werden. (Garbage Collection!)

Nun gilt: Für jede Belegung der Variablen X, die alle Klausel in F erfüllt, gibt es mindestens ein Matching $M \subseteq W$ mit genau $2 \cdot i \cdot j$ Tripel.

Jedes Machting $M \subseteq W$ mit genau $2 \cdot i \cdot j$ Tripel definiert eine Belegung der Variablen in X und ein unter dieser Belegung wahres Literal für jede Klausel in F.

20.6 Zahlenprobleme

Problem: KNAPSACK
Gegeben: Gewichte $g_1, \ldots, g_n \in \mathbb{N}$, Nutzwerte $a_1, \ldots, a_n \in \mathbb{N}$, eine Gewichtsschranke $G \in \mathbb{N}$ und eine Nutzschranke $A \in \mathbb{N}$.
Frage: Gibt es eine Indexmenge $R \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in R} g_i \leq G$ und $\sum_{i \in R} a_i \geq A$?
Satz: KNAPSACK ist NP-vollständig.

1. KNAPSACK $\in$ NP: Rate und verifiziere.

2. KNAPSACK ist NP-schwer:

   Wir zeigen: 3SAT $\leq_p$ KNAPSACK

   Sei $F = \{C_1, \ldots, C_m\}$ mit $C_i = \{C_{i,1}, C_{i,2}, C_{i,3}\} \subseteq \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$ eine Instanz für 3SAT.

   Setze $A := G := \underbrace{44\ldots4}_{m}\underbrace{11\ldots1}_{n}$ (= eine $(m+n)$-stellige Dezimalzahl).

Wir konstruieren $2n + 2m$ Gewichte und Nutzwerte mit $a_i = g_i$.

Aufbau der Nutzwerte (Gewichte) in der Dezimaldarstellung:

Klauseln				Variable				Literale
c_1	c_2	$\dots$	c_m	x_1	x_2	$\dots$	x_n	
0	1	$\dots$	0	1	0	$\dots$	0	x_1
0	0	$\dots$	0	1	0	$\dots$	0	$\overline{x_1}$
		$\vdots$				$\vdots$		$\vdots$
1	1	$\dots$	0	0	0	$\dots$	1	$\overline{x_m}$

Ergänzungswerte

1	0	$\dots$	0	0	0	$\dots$	0
0	1	$\dots$	0	0	0	$\dots$	0
		$\ddots$				$\ddots$	
0	0	$\dots$	1	0	0	$\dots$	0
2	0	$\dots$	0	0	0	$\dots$	0
0	2	$\dots$	0	0	0	$\dots$	0
		$\ddots$				$\ddots$	
0	0	$\dots$	2	0	0	$\dots$	0

Wenn es eine Teilmenge $R \subseteq \{1, \dots, 2m+2n\}$ gibt mit $\sum_{i \in R} a_i = A$, dann ist dadurch eindeutig eine Belegung der Variablen $x_1, \dots, x_n$ gegeben, die jede Klausel erfüllt (und umgekehrt). $\square$

Beispiel:
$F = \{C_1, C_2, C_3\},\ X = \{x_1, x_2, x_3, x_4\} \Rightarrow m = 3, n = 4$
$C_1 = \{x_1, \overline{x_2}, x_3\},\ C_2 = \{\overline{x_1}, x_2, \overline{x_4}\},\ C_3 = \{\overline{x_1}, \overline{x_2}, \overline{x_3}\}$
$A = G = 4441111$

$$
\begin{array}{lcllll}
a_1 & = & g_1 & = & 1\,0\,0 & 1\,0\,0\,0 \quad (x_1) \\
a_2 & = & g_2 & = & 0\,1\,1 & 1\,0\,0\,0 \quad (\overline{x_1}) \\
a_3 & = & g_3 & = & 0\,1\,0 & 0\,1\,0\,0 \quad (x_2) \\
a_4 & = & g_4 & = & 1\,0\,1 & 0\,1\,0\,0 \quad (\overline{x_2}) \\
a_5 & = & g_5 & = & 1\,0\,0 & 0\,0\,1\,0 \quad (x_3) \\
a_6 & = & g_6 & = & 0\,0\,1 & 0\,0\,1\,0 \quad (\overline{x_3}) \\
a_7 & = & g_7 & = & 0\,0\,0 & 0\,0\,0\,1 \quad (x_4) \\
a_8 & = & g_8 & = & 0\,1\,0 & 0\,0\,0\,1 \quad (\overline{x_4})
\end{array}
$$

Ergänzungswerte

$$
\begin{array}{lcllll}
a_9 & = & g_9 & = & 1\,0\,0 & 0\,0\,0\,0 \\
a_{10} & = & g_{10} & = & 0\,1\,0 & 0\,0\,0\,0 \\
a_{11} & = & g_{11} & = & 0\,0\,1 & 0\,0\,0\,0 \\
a_{12} & = & g_{12} & = & 2\,0\,0 & 0\,0\,0\,0 \\
a_{13} & = & g_{13} & = & 0\,2\,0 & 0\,0\,0\,0 \\
a_{14} & = & g_{14} & = & 0\,0\,2 & 0\,0\,0\,0
\end{array}
$$

Lösungsmenge $R = \{2, 3, 5, 7\}$

$$
\begin{array}{lll}
 & 0\,1\,1 \quad 1\,0\,0\,0 & \\
 & 0\,1\,0 \quad 0\,1\,0\,0 & \\
 & 1\,0\,0 \quad 0\,0\,1\,0 & \\
+ & 0\,0\,0 \quad 0\,0\,0\,1 & \\
\hline
= & 1\,2\,1 \quad 1\,1\,1\,1 & (\sum_{i\in R} a_i) \\
+ & 1\,0\,0 \quad 0\,0\,0\,0 & \\
+ & 2\,0\,0 \quad 0\,0\,0\,0 & \\
+ & 0\,2\,0 \quad 0\,0\,0\,0 & \\
+ & 0\,0\,1 \quad 0\,0\,0\,0 & \\
+ & 0\,0\,2 \quad 0\,0\,0\,0 & \\
= & 4\,4\,4 \quad 1\,1\,1\,1 &
\end{array}
$$

Ein einfacher Algorithmus für das KNAPSACK Problem:
Gegeben: Gewichte $g_1, \ldots, g_n \in \mathbb{N}$, Nutzwerte $a_1, \ldots, a_n \in \mathbb{N}$ und eine Gewichtsschranke $G \in \mathbb{N}$.
Aufgabe: Berechne den maximalen Nutzen $A_{\max}$, d.h. den maximalen Wert $A_{\max}$ für den es eine Menge $R \subseteq \{1, \ldots, n\}$ gibt mit

$$\sum_{i\in R} g_i \leq G \text{ und } \sum_{i\in R} a_i = A_{\max}.$$

Sei B ein Array mit $G+1$ Einträgen $B[0], \ldots, B[G]$ initialisiert mit -1.

$B[0] = 0;$
for $(j = 1;\ j \leq n;\ j++)$
    for $(i = G;\ i \geq 0;\ i--)$
        if $(i + g_j \leq G)$ AND $B[i + g_j] \geq 0$ AND $(B[i + g_j] < B[i] + a_j)$
            $B[i + g_j] = B[i] + a_j;$

Ausgabe: $max\{B[0], \ldots, B[G]\}$
Laufzeit: $O(n \cdot G)$
Bemerkung: Der obige Algorithmus hat keine polynomielle sondern eine exponentielle Laufzeit für KNAPSACK, da die Eingabe die Größe

$$\sum_{i=1}^{n}(log_2(a_i) + log_2(g_i)) + log_2(A) + log_2(G)$$

hat.
Beispiel: $a_i : 7, 10, 20, 15, 8$, $g_i : 1, 4, 2, 3, 4$, $G = 8$

	0	1	2	3	4	5	6	7	8
	0	-1	-1	-1	-1	-1	-1	-1	-1
$j = 1$	0	7	-1	-1	-1	-1	-1	-1	-1
$j = 2$	0	7	-1	-1	10	17	-1	-1	-1
$j = 3$	0	7	20	27	10	17	30	37	-1
$j = 4$	0	7	20	27	22	35	42	37	32
$j = 5$	0	7	20	27	22	35	42	37	32

$A_{\max} = B[6] = 42$
Definition: Sei I die Eingabe eines Problems π, sei $L(I)$ die Länge der Eingabe I (in der üblichen Darstellung; Zahlen binär kodiert) und sei MAX(I) die Größe der größten in I vorkommenden Zahl.
Ein Problem π heißt *Zahlenproblem*, wenn sich MAX(I) nicht durch ein Polynom in $L(I)$ beschränken läßt. (wenn eine Beschränkung von MAX(I) das Problem enschränkt.)
Bemerkung: KNAPSACK ist eine Zahlenproblem.
Keine Zahlenprobleme sind zum Beispiel: CLIQUE, INDEPENDENT SET, VERTEX COVER, SAT, 3SAT, DHC
Definition: Ein Algorithmus für ein Zahlenproblem π heißt *pseudopolynomiell*, wenn die Rechenzeit durch ein Polynom in $L(I) + \text{MAX}(I)$ beschränkt ist.
Definition: Für ein Problem π und ein Polynom p sei π_p das Teilproblem von π, bei dem nur die Eingaben I mit $\text{MAX}(I) \leq p(L(I))$ betrachtet werden.
Ein Problem π heißt *stark* NP-vollständig (NP-vollständig im *starken Sinne*), wenn es ein Polynom p gibt, so daß selbst das Teilproblem π_p NP-vollständig ist.

Ein NP-vollständiges Problem, dass nicht stark NP-vollständig ist, heißt *schwach* NP-vollständig (NP-vollständig im *schwachen Sinne*).
Bemerkung: π_p ist kein Zahlenproblem.
Bemerkung: NP-vollständige Probleme, die keine Zahlenprobleme sind, sind immer stark NP-vollständig.
Satz: Falls P $\neq$ NP, dann gibt es für stark NP-vollständige Probleme π keine pseudopolynomiellen Algorithmen.
Beweis: Jeder pseudopolynomielle Algorithmus für π ist ein polynomieller Algorithmus für π_p und somit wäre das NP-vollständige problem π_p in polynomieller Zeit entscheidbar . $\square$
Bemerkung: KNAPSACK ist nicht stark NP-vollständig (sondern schwach NP-vollständig).
Problem: 3-PARTITION
Gegeben: Zahlen $a_1, \ldots, a_{3n} \in \mathbb{N}$.
Frage: Kann man $\{1, \ldots, 3n\}$ disjunkt in 3-elementige Mengen $S_1, \ldots, S_n$ aufteilen, so daß

$$\forall i = 1, \ldots, n : \sum_{j \in S_i} a_j = B = \frac{\sum_{i=1}^{3n} a_i}{n}?$$

Satz: 3-PARTITION ist stark NP-vollständig.
Beweis: 3-DM $\leq_p$ 4-PARTITION $\leq_p$ 3-PARTITION (siehe Garey/Johnson)
Beispiel:
Gegeben: $a_1 = 1$, $a_2 = 2$, $a_3 = 2$, $a_4 = 3$, $a_5 = 8$, $a_6 = 3$, $a_7 = 2$, $a_8 = 9$, $a_9 = 1$, $a_{10} = 7$, $a_{11} = 6$, $a_{12} = 4$ $3n = 12 \Rightarrow n = 4$, $\sum_{i=1}^{3n} a_i = 48$, $B = \frac{48}{n} = 12$
Lösung:
$S_1 = \{2, 9, 1\},\ S_2 = \{8, 3, 1\},\ S_3 = \{7, 3, 2\},\ S_4 = \{6, 4, 2\}$
Bemerkung: 3-PARTITION ist auch dann stark NP-vollständig, wenn alle a_i der Eingabe größer als $\frac{B}{4}$ und kleiner als $\frac{B}{2}$ sind. Konstruiere aus der Instanz $a_1, \ldots, a_{3n}$ einfach eine neue Instanz $b_1, \ldots, b_{3n}$ mit $b_i = B + a_i$ ($B = \frac{\sum_{i=1}^{3n} a_i}{n}$). Daraus folgt, dass jede Menge, dessen Summe B ergibt, zwangsläufig aus genau drei Werten bestehen muss.
Bemerkung: Stark NP-vollständige Probleme bleiben NP-vollständig, auch wenn die Zahlen in den Instanzen unär (zur Basis 1) kodiert vorliegen.
Anmerkung zur 3-PARTITION: 3-Partition ist kein Zahlenproblem obwohl jede Instanz nur aus Zahlen besteht.
Bemerkung: Mit Hilfe von Entscheidungsprozeduren kann oft explizit eine Lösung konstruiert werden.
Wenn es einen Algorithmus A gibt, der zum Beispiel das Cliquen-Problem in polynomieller Zeit löst, dann kann man auch in polynomieller Zeit eine maximale

Clique finden.
Idee: Teste für jeden Knoten u nacheinander, ob $G-u$ noch eine Clique der Größe k hat. Falls ja, entferne u und wiederhole den Schritt solange bis eine Clique der Größe k übrig bleibt.
Bemerkungen: Die Konstruktion einer maximalen Clique ist nicht in polynomieller Zeit möglich, falls P $\neq$ NP.
Definition: Die Klasse co-NP ist die Menge alle Sprachen $L \subseteq \Sigma^*$ mit $\Sigma^* - L \in NP$.
Beispiel:
Sei L die Menge aller ungerichteten Graphen, die keinen Hamilton-Kreis enthalten. Dann ist $L \in$ co-NP.
Korollar: Wenn das Komplement $\overline{L} = \Sigma^* - L$ eines NP-vollständiges Problems L in NP ist, dann gilt:

$$\text{co-NP} = \text{NP}.$$

Beweis:
Wenn L NP-schwer ist, dann gilt $\forall L' \in \text{NP} \ : \ L' \leq_p L$ und aufgrund der Definition der Reduktion $\overline{L'} \leq_p \overline{L}$. Wenn nun $\overline{L} \in$ NP gilt, dann sind auch alle Sprachen $\overline{L'} \in$ co-NP in NP, und somit ist co-NP = NP. $\square$
Bemerkung: Da co-NP $\neq$ NP angenommen wird, sind die Komplementprobleme der NP-vollständigen Probleme nicht in NP. (Suchen sie keine NP-Vollständigkeitsbeweise für L, wenn $\overline{L}$ in NP liegt.)
Beispiel:
Gegeben: Eine Zahl $n \in \mathbb{N}$.
Frage: Ist n eine Primzahl?
Das Komplementproblem, ob eine Zahl n keine Primzahl ist, liegt offensichtich in NP (rate einen Zahl k, $1 < k < n$, und teste, ob k ein Teiler von n ist.) Also ist das Problem, ob eine Zahl n eine Primzahl ist, nicht NP-vollständig, falls NP $\neq$ co-NP.

20.7 Approximationsalgorithmen

Definition: Ein *Optimierungsproblem* (Minimierungs- bzw. Maximierungsproblem) π ist ein Problem, dessen Antwort aus einem optimalen (minimalen bzw. maximalen) Lösungswert besteht.
Sei

- D_π die Menge aller zulässigen Eingaben (Instanzen),
- $S_\pi(I)$ die Menge aller Lösungen für eine Instanz $I \in D_\pi$ und
- f eine Funktion, die jeder Lösung $s \in S_\pi(I)$ einen (positiven) Wert $f(I, s)$ zuordnet.

Für eine Instanz I bezeichnet $\mathrm{OPT}(I)$ den Wert einer optimalen Lösung (d.h. einer Lösung mit minimalem bzw. maximalen Wert).
Ein *Approximationsalgorithmus* A für ein Optimierungsproblem π liefert für jede Instanz $I \in D_\pi$ einen Lösungswert $A(I)$.
Die *Güte* der Lösung $A(I)$ ist

$$R_A(I) := \begin{cases} \frac{A(I)}{\mathrm{OPT}(I)} & \text{für Minimierungsprobleme} \\ \\ \frac{\mathrm{OPT}(I)}{A(I)} & \text{für Maximierungsprobleme} \end{cases}$$

Die *worst-case Güte* von A ist

$$R_A := \sup\{R_A(I) \ : \ I \in D_\pi\} = \inf\{r \geq 1 \ : \ R_A(I) \leq r, \ \forall I \in D_\pi\}$$

Die *asymptotische worst-case Güte* von A ist

$$R_A^\infty := \inf\{r \geq 1 \ : \ \exists n : \forall I \in D_\pi : (\mathrm{OPT}(I) \geq n \Rightarrow R_A(I) \leq r)\}$$

Problem: BIN PACKING
Gegeben: Endliche Menge $U = \{o_1, \ldots, o_n\}$ von Objekten mit Größen $g(o_i) \in \{0, 1\}$.
Lösung: Eine Partition von U in disjunkte Mengen $U_1, \ldots, U_k$, so daß die Summen der Größen der Objekte in den Mengen nicht größer als 1 ist.
Wert der Lösung: Anzahl k der Mengen.
Optimierungsrichtung: Minimierung
Definition: Ein NP-*Optimierungsproblem* ist ein Optimierungsproblem für das zusätzlich folgende Eigenschaften gilt.

1. $s \in S_\pi(I)$ ist in polynomieller Zeit entscheidbar.
2. $f(I, s)$ ist in polynomieller Zeit berechenbar.

Beispiel:

1. π = BIN PACKING;
2. I = Menge von Objekten $U = \{o_1, \ldots, o_n\}$ mit Größen $g(o_i) \in [0, 1]$.
3. $S_\pi(I)$ = Menge aller vollständigen disjunkten Partitionen von U, so daß die Summe der Größen der Objekte in jeder Menge der Partitionen kleiner gleich 1 ist.
4. $f(I, s) = k$ für eine Lösung (Partition) s mit k Mengen
5. Minimierungsproblem

Satz: BIN PACKING ist stark NP-vollständig.
Bemerkung: BIN PACKING ist ein Optimierungsproblem (Minimierungsproblem). Wir nennen Optimierungsprobleme NP-vollständig, wenn das entsprechende Entscheidungsproblem für das Erreichen eines Maximal-/Minimalwertes NP-vollständig ist.
Entscheidungsproblem: BIN PACKING
Gegeben: Endliche Menge $U = \{o_1, \ldots, o_n\}$ von Objekten mit Größen $g(o_i) \in [0, 1]$ und eine Zahl k.
Frage: Gibt es eine Partition von U in höchstens k disjunkte Mengen $U_1, \ldots, U_k$, so daß die Summen der Größen der Objekte in den Mengen nicht größer als 1 ist.
Beweis:

1. BIN PACKING ist in NP, rate und verifiziere.
2. Reduktion von 3-PARTITION (stark NP-vollständig)

   Sei $a_1, \ldots, a_{3n} \in \mathbb{N}$ eine Instanz für 3-Partition und $B = \frac{\sum_{i=1}^{3n} a_i}{n}$.

   Ohne Beschränkung der Allgemeinheit gilt für alle a_i, $\frac{B}{4} < a_i < \frac{B}{2}$. (Der Wert B kann nur aus der Summe von genau drei Zahlen gebildet werden.)

   Sei $U = \{o_1, \ldots, o_{3n}\}$ eine Menge von Objekten mit Größen $g(o_i) = \frac{a_i}{B}$ und sei $k = n$.

   Diese Instanz besitzt genau dann für das Entscheidungsproblem BIN PACKING eine Lösung, wenn $a_1, \ldots, a_{3n}$ eine Lösung für 3-Partition besitzt. □

Ein Algorithmus (FIRST FIT) für BIN PACKING:

1. Starte mit einer Folge von n leeren Behältern $B_1, B_2, \ldots, B_n$.
2. Bearbeite die Objekte in der Reihenfolge $o_1, \ldots, o_n$ wie folgt:

   Plaziere das nächste Objekt o_i in den Behälter mit dem kleinsten Index j, in den es noch hineinpaßt.

Sei $\mathrm{FF}(I)$ der Wert der Lösung, die Algorithmus FIRST-FIT für eine Eingabe I berechnet.
Dann gilt:

$$\mathrm{FF}(I) \leq \left\lceil 2 \cdot \sum_{i=1}^{n} g(o_i) \right\rceil .$$

Es gibt höchstens einen Behälter, der weniger als bis zur Hälfte gefüllt ist.
Da $\mathrm{OPT}(I) \geq \lceil \sum_{i=1}^{n} g(o_i) \rceil$, folgt $\mathrm{FF}(I) \leq 2 \cdot \mathrm{OPT}(I)$ und $R_{\mathrm{FF}}^{\infty} \leq R_{\mathrm{FF}} \leq 2$.
(Mehr ist so nicht zu sehen.)
Bemerkungen:

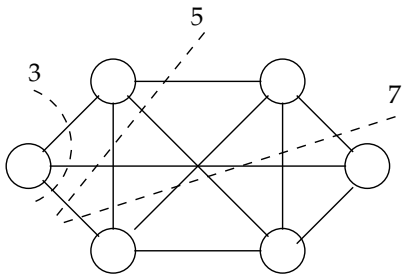
1. $\forall\ I \in D_\pi : \quad \mathrm{FF}(I) \leq \frac{17}{10}\mathrm{OPT}(I) + 2$,

2. Es existieren Instanzen I mit beliebig großem $\mathrm{OPT}(I)$, so daß

$$\mathrm{FF}(I) \geq \frac{17}{10}(\mathrm{OPT}(I) - 1).$$

3. Daraus folgt, dass $R_{\mathrm{FF}}^{\infty} = \frac{17}{10}$.

Problem: MAXIMUM CUT
Gegeben: Ein Graph $G = (V, E)$.
Lösung: Eine Knotenmenge $S \subseteq V$.
Wert der Lösung: Anzahl der Kanten, die einen Endknoten in S und einen Endknoten in $V - S$ haben.
Optimierungsrichtung: Maximierung
Bemerkung: MAXIMUM CUT (Entscheidungsversion) ist NP-vollständig.
Ein einfacher Approximationsalgorithmus A für MAXIMUM CUT (lokale Verbesserung):
Starte mit $S = \emptyset$. Füge einen Knoten zu S hinzu oder nehme einen Knoten aus S heraus, falls sich die Lösung dadurch verbessern läßt.
Der Algorithmus terminiert nach höchstens $|E|$ Schritten, da die Lösung nur höchstens $|E|$ mal verbessert werden kann.

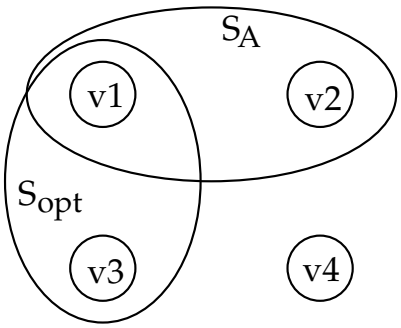


Satz: $R_A \leq 2$
Beweis: Betrachte eine disjunkte Zerlegung der Knotenmenge in vier Mengen $V_1, V_2, V_3, V_4 \subseteq V$, so daß der Algorithmus A die Knotenmenge $S = V_1 \cup V_2$ berechnet und $S = V_1 \cup V_3$ eine optionale Lösung ist.
Sei $e_{i,j}$ die Anzahl der Kanten, die einen Knoten in V_i und den anderen Knoten in V_j haben.
Für jeden Knoten $u \in V_1$ gilt

$$\begin{aligned} &|\{v \in V_1 \mid \{u, v\} \in E\}| + |\{v \in V_2 \mid \{u, v\} \in E\}| \\ &\leq |\{v \in V_3 \mid \{u, v\} \in E\}| + |\{v \in V_4 \mid \{u, v\} \in E\}|. \end{aligned}$$

Summation über alle $u \in V_1$ ergibt $2 \cdot e_{1,1} + e_{1,2} \leq e_{1,3} + e_{1,4}$ und somit

$$e_{1,2} \leq e_{1,3} + e_{1,4}.$$



Aus den anderen Mengen erhält man die Ungleichungen

- $e_{1,2} \leq e_{2,3} + e_{2,4}$
- $e_{3,4} \leq e_{2,3} + e_{1,3}$
- $e_{3,4} \leq e_{1,4} + e_{2,4}$

Die Addition aller vier Ungleichungen, eine anschließende Division durch 2, und die Addition der Ungleichung $e_{1,4} + e_{2,3} \leq e_{1,4} + e_{2,3} + e_{1,3} + e_{2,4}$ ergibt

$$\underbrace{e_{1,2} + e_{3,4} + e_{1,4} + e_{2,3}}_{\mathrm{OPT}(I)} \leq 2 \cdot \underbrace{(e_{1,3} + e_{1,4} + e_{2,3} + e_{2,4})}_{A(I)}$$

und somit $R_A \leq 2$. $\square$

Definition: Ein Approximationsalgorithmus A für ein Optimierungsproblem π ist ein *Approximationsalgorihmus mit additivem Fehler*, wenn für eine Konstante c und für alle $I \in D_\pi$ gilt: $|A(I) - \mathrm{OPT}(I)| \leq c$.

Satz: Falls P $\neq$ NP, dann existiert kein polynomieller Approximationsalgorithmus mit additivem Fehler für KNAPSACK (Maximierung des Nutzwerts).

Beweis: Angenommen es gibt einen solchen Algorithmus A mit additivem Fehler k. Sei I eine Instanz für die Entscheidungsversion von KNAPSACK mit Nutzschranke B. Multipliziere alle Gewichte und Nutzwerte mit $k + 1$.

Die neue Instanz I' hat genau dann eine Lösung für KNAPSACK, wenn I eine Lösung für KNAPSACK hat. Da sich verschiedene Werte der Lösungen von I' mindestens um $k + 1$ unterscheiden bestimmt A für I' eine optimale Lösung. $\square$

Frage: Gibt es für das INDEPENDENT SET Problem einen polynomiellen Approximationsalgorithmus A mit additivem Fehler, falls P $\neq$ NP?

Antwort: Nein! Angenommen es gibt einen Algorithmus A mit additivem Fehler k. Sei G' die disjunkte Vereinigung von $(k + 1)$ Kopien eines Graphen G.

Wenn G ein Independent Set der Größe r hat, dann hat G' ein Independent Set der Größe $(k + 1) \cdot r$. Wegen

$$|A(G') - \mathrm{OPT}(G')| = |A(G') - (k + 1)\mathrm{OPT}(G)| \leq k$$

müßte A in mindestens einer Kopie ein Independent Set der Größe $\mathrm{OPT}(G)$ liefern.

Problem: GRAPH k-COLORABILTY

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl k, $1 \leq k \leq |V|$.

Frage: Gibt es eine vollständige disjunkte Aufteilung der Knoten in k unabhängige Mengen (INDEPENDENT SETS) (eine Färbung $f : V \to \{1, \ldots, k\}$ der Knoten mit k Farben), so dass zwei adjazente Knoten nicht in der gleichen Menge liegen (nicht die gleiche Farbe haben, $\forall\{u, v\} \in E : f(u) \neq f(v)$).
Bemerkung: GRAPH-3-COLORABILITY ist NP-vollständig.
Problem: CHROMATIC NUMBER
Gegeben: Ein ungerichteter Graph $G = (V, E)$.
Lösung: Eine vollständige disjunkte Aufteilung der Knoten in k Mengen .
Wert der Lösung: Anzahl der Mengen.
Optimierungsrichtung: Minimierung
Satz: Falls P $\neq$ NP, dann gibt es keinen polynomiellen Approximations-Algorithmus A für CHROMATIC NUMBER mit $R_A < \frac{4}{3}$.
Beweis: Ein solcher Algorithmus A würde jeden 3-färbbaren Graphen mit höchstens 3 Farben färben. $\square$
Problem: TRAVELLING SALESMAN PROBLEM (TSP)
Gegeben: Eine $n \times n$ Matrix d mit nichtnegativen Einträgen (Distanzen).
Lösung: Eine Bijektion $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.
Wert der Lösung: $d(\pi(1), \pi(2)) + d(\pi(2), \pi(3)) + \ldots + d(\pi(m-1), \pi(i_m)) + d(\pi(m), \pi(1))$.
Optimierungsrichtung: Minimierung
Satz: Falls P $\neq$ NP, dann gibt es keinen polynomiellen Approximations-Algorithmus A für TSP mit $R_A < \infty$.
Beweis: Angenommen ein solcher Algorithmus A mit $R_A = k < \infty$ existiert. Sei $G = (V, E)$ ein ungerichteter Graph. Betrachte die folgende Instanz I für das TSP.

$$d(u, v) := \begin{cases} 1, & \{u, v\} \in E \\ k \cdot |V| & \{u, v\} \notin E \end{cases}$$

Nun gilt folgendes:
Wenn G einen Hamiltonkreis besitzt, dann ist $\mathrm{OPT}(I) = |V|$, ansonsten gilt $\mathrm{OPT}(I) > k \cdot |V|$. Mit Hilfe von A könnten wir nun entscheiden, ob G einen Hamiltonkreis hat oder nicht. $\square$

Bermerkung: Der obige Satz gilt nicht für TSPs, die die Dreiecksungleichungen erfüllen.
Dreiecksungleichungen: $\forall i, j, k \in \{1, \ldots, n\} : d(i, k) \leq d(i, j) + d(j, k)$
Definition:

1. Ein *Approximationsschema* für ein Optimierungsproblem π ist ein Algorithmus A, der zu jedem $\epsilon > 0$ und zu jeder Instanz $I \in D_\pi$ einen Lösungswert $A(I, \epsilon)$ berechnet, deren Güte durch $1 + \epsilon$ beschränkt ist.

2. Algorithmus A ist ein *polynomielles Approximationsschema*, wenn die Laufzeit von A für jedes konstante ϵ polynomiell in der Länge der Instanz $L(I)$ ist.

3. Algorithmus A ist ein voll polynomielles Approximationsschema, wenn die Laufzeit von A polynomiell in $L(I)$ und $\frac{1}{\epsilon}$ ist.

Satz: Für jedes $\epsilon > 0$ gibt einen polynomiellen Approximationsalgorithmus A für das KNAPSACK Problem mit einer Güte $R_A \leq 1 + \epsilon$.
Beweis: Sei $I = (g_1, \ldots, g_n, G, a_1, \ldots, a_n)$ eine Instanz für KNAPSACK.
Wir suchen eine Teilmenge $S \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in S} g_i \leq G$, so daß $\sum_{i \in S} a_i$ maximal ist.
Sei $V := \max\{a_1, \ldots, a_n\}$. Für $i = 0, \ldots, n$, $v = 0, \ldots, nV$ sei $w(i, v)$ das minimale Gewicht, das mit einer Auswahl aus den ersten i Objekten erreicht werden kann, wenn der Nutzwert genau v ist.
Algorithmus:

1. Initialisiere $w(0, 0) = 0$ und $w(i, v) = \infty$ für alle anderen i, v;

2. Berechne $w(i + 1, v) = \min\{w(i, v), w(i, v - a_{i+1}) + g_{i+1}\}$;

3. Bestimme das größte v mit $w(n, v) \leq G$;

Algorithmus A berechnet die optimale Lösung in Zeit $O(n^2 V)$.
Definiere für ein $b \in \mathbb{N}_0$ die Instanz $I' = (g_1, \ldots, g_n, G, a'_1, \ldots, a'_n)$ mit

$$a'_i = 2^b \cdot \left\lfloor \frac{a_i}{2^b} \right\rfloor$$

(setze die b niederwertigen Bits von a_i auf 0)
Der Algorithmus benötigt nun $O(n^2 \cdot \frac{V}{2^b})$ Rechenzeit. (Teile alle a'_i durch 2^b und multipliziere den berechneten maximalen Nutzwert mit 2^b.)
Die Güte der Lösung für I': (S optimal für I, S' optimal für I')

$$\sum_{i \in S} a_i \geq \sum_{i \in S'} a_i \geq \sum_{i \in S'} a'_i \geq \sum_{i \in S} a'_i \geq \sum_{i \in S} (a_i - 2^b) \geq (\sum_{i \in S} a_i) - n \cdot 2^b$$

Die Lösung ist höchstens um $n \cdot 2^b$ kleiner als die optimale Lösung.

$$A(I) \geq \mathrm{OPT}(I) - n \cdot 2^b$$

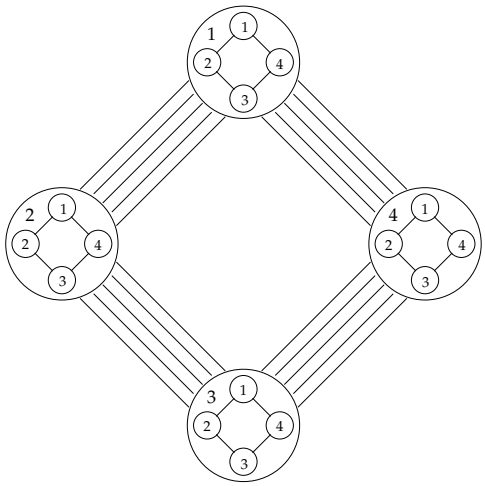
$$R_A = \frac{\mathrm{OPT}(I)}{A(I)} \leq 1 + \frac{n \cdot 2^b}{A(I)} \leq 1 + \frac{n \cdot 2^b}{V}$$

denn $V = \max\{a_1, \ldots, a_n\}$ ist eine untere Schranke für jedes $A(I)$.

Für jedes $\epsilon > 0$ schneide die letzten $b = \lfloor \log \frac{\epsilon V}{n} \rfloor$ Stellen der Nutzwerte ab.
Dann ist A ein Approximationsalgorithmus mit $R_A \leq 1 + \epsilon$ und Laufzeit $O(\frac{n^3}{\epsilon})$.
A ist ein voll polynomielles Approximationsschema. □

MAXIMUM INDEPENDENT SET für $G = (V, E)$

Sei G^2 der Graph mit Knotenmenge $V \times V$ und Kantenmenge
$\{ \{(u, u'), (v, v')\} |$ entweder ist $u = v$ und $\{u', v'\} \in E$ oder $u' = v'$ und $\{u, v\} \in E \}$



Es gilt $\mathrm{OPT}(G^2) = (\mathrm{OPT}(G))^2$.
Wenn G ein Independent Set I der Größe k hat, dann hat G^2 das Independent Set $I^2 = \{(u, v) \ : \ u, v \in I\}$ der Größe k^2.
Wenn I^2 ein Independent Set von G^2 der Größe k^2 ist, dann sind $I = \{u \ : \ (u, v) \in I^2\}$ und $I_u = \{v \ : \ (u, v) \in I^2\}$, $u \in I$ Independent Sets für G. I oder wenigstens ein I_u muß mindestens k Knoten enthalten, d.h. G hat ein Independent Set der Größe k. Dieses läßt sich leicht aus I^2 bestimmen.
Satz: Wenn es einen polynomiellen Approximationsalgorithmus A für INDEPENDENT SET mit $R_A^\infty < \infty$ gibt, dann gibt es auch ein polynomielles Approximationsschema für INDEPENDENT SET.
Beweis: Sei $O(n^k)$ die Laufzeit von A auf Eingabe G mit n Knoten.
Dann ist $O(n^{2k})$ die Laufzeit von A auf G^2.
Wegen $R_A^\infty < \infty$ und da man annehmen kann, dass A ein Independent Set konstanter Größe in polynomieller Zeit findet, muß auch $R_A := g < \infty$ gelten.
Für G^2 erhält man eine Lösung I^2 der Größe $\frac{\mathrm{OPT}(G)^2}{g}$, woraus eine Lösung für G der Größe $\frac{\mathrm{OPT}(G)}{g^{\frac{1}{2}}}$ konstruiert werden kann.
Durch Wiederholen obiger Prozedur erhält man einen Approximationsalgorithmus mit einer Laufzeit von $O(n^{2^i k})$ und einer Güte von $g^{\frac{1}{2^i}}$. □

20.8 Weitere Komplexitätsklassen

Die Struktur von NP und co-NP
Die Klasse co-NP ist die Menge alle Sprachen $L \subseteq \Sigma^*$ mit $\Sigma^* - L \in NP$.
Da viele Probleme aus co-NP anscheinend nicht in NP sind, könnte man vermuten, dass co-NP $\neq$ NP gilt. Diese Vermutung ist jedoch stärker als die Vermutung P $\neq$ NP.
Da P = co-P gilt, würde NP $\neq$ co-NP die Aussage P $\neq$ NP implizieren, obwohl es sein könnte, dass P $\neq$ NP und NP = co-NP ist.
Dennoch gibt es eine starke Beziehung zwischen NP-vollständigen Problemen und der Vermutung NP $\neq$ co-NP. (Wenn das Komplement eines NP-vollständigen Problemns NP-vollständig ist, dann gilt NP = co-NP.)

20.9 Aufzählungsprobleme

Definition: Sei π ein Optimierungsproblem.

1. D_π = Menge der Instanzen
2. $S_\pi(I)$ = Menge der Lösungen für I
3. Das Aufzählungsproblem für π ist die Berechnung von $|S_\pi(I)|$ für eine Instanz $I \in D_\pi$.

Bemerkung: Die Lösungen brauchen nicht explizit angegeben werden.
Definition: Ein Aufzählungsproblem π ist in der Klasse #P (Number P), wenn es eine pnichtdeterministische Turingmaschine M gibt, so daß für jede Instanz $I \in D_\pi$ die Anzahl der akzeptierenden Berechnungen für I genau $|S_\pi(I)|$ ist und die Länge der akzeptierenden Berechnungen polynomiell in der Instanz I ist.
Bemerkung: Ist für jedes $I \in D_\pi$ und jedes $s \in S_\pi(I)$ die Länge von s polynomiell in der Länge von I und ist $s \in S_\pi(I)$ für ein gegebenes I und s in polynomieller Zeit entscheidbar, dann ist $\pi \in \#P$.
Definition: Ein Aufzählungsproblem π heißt *parsimonious* transformierbar auf ein Aufzählungsproblem π', wenn es eine deterministische, polynomielle Turingmaschine M gibt, die eine Funktion $f : D_\pi \to D_{\pi'}$ berechnet, so daß

$$\forall\ I \in D_\pi :\ |S_\pi(I)| = |S_{\pi'}(f(I))|.$$

Definition: Ein Aufzählungsproblem π heißt #P-vollständig, wenn es in #P ist und jedes $\pi' \in$ #P auf π parsimonious transformierbar ist.
Satz: Die Berechnung der Anzahl der erfüllenden Belegungen für eine gegebene SAT-Instanz ist #P-vollständig.
Satz: Die Berechnung der Anzahl der Hamilton-Kreise in einem gerichteten Graphen ist #P-vollständig.

Beweis:

1. $\pi \in$ #P ist offensichtlich.
2. Unsere Reduktion SAT $\leq_p$ DHC ist bereits parsimonious.

Bemerkung: Es gibt #P-vollständige Aufzählungsprobleme, obwohl das unterliegende Entscheidungsproblem effizient in polynomieller Zeit entscheidbar ist.
Beispiel: Das Aufzählungsproblem für die Anzahl der verschiedenen perfekten Matchings in einem bipartiten Graphen ist #P-vollständig. (Ohne Beweis!)

20.10 Die Komplexitätsklasse PSPACE

Weitere Komplexitätsklassen:

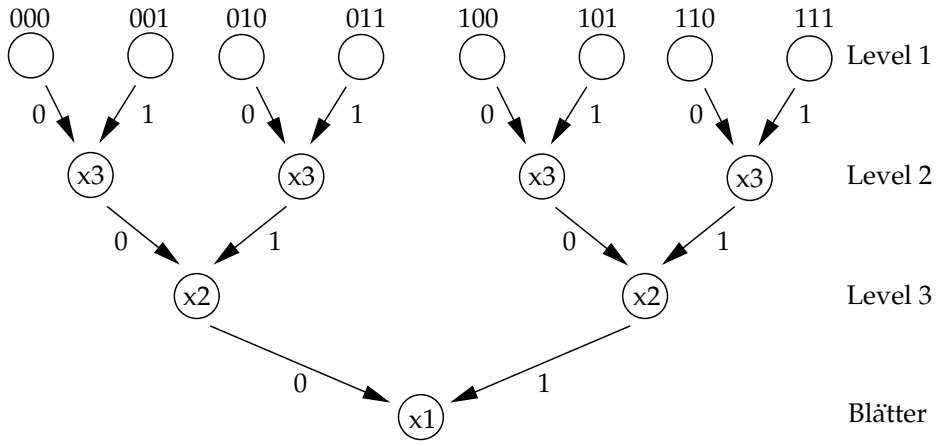
- $\mathrm{P} = \mathrm{TIME}(n^k) = \bigcup_{j>0} \mathrm{TIME}(n^j)$
- $\mathrm{NP} = \mathrm{NTIME}(n^k) = \bigcup_{j>0} \mathrm{NTIME}(n^j)$
- $\mathrm{L} = \mathrm{SPACE}(log(n))$
- $\mathrm{NL} = \mathrm{NSPACE}(log(n))$
- $\mathrm{EXP} = \mathrm{EXPTIME} = \mathrm{TIME}(2^{n^k}) = \bigcup_{j>0} \mathrm{TIME}(2^{n^j})$
- $\mathrm{NEXP} = \mathrm{NEXPTIME} = \mathrm{NTIME}(2^{n^k}) = \bigcup_{j>0} \mathrm{NTIME}(2^{n^j})$
- $\mathrm{PSPACE} = \mathrm{SPACE}(n^k)$
- $\mathrm{NPSPACE} = \mathrm{NSPACE}(n^k)$
- Platz-Hierarchiesatz: $\mathrm{L} \subsetneq \mathrm{PSPACE}$
- Zeit-Hierarchiesatz: $\mathrm{P} \subsetneq \mathrm{EXP}$
- $\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP}$

Definition: Ein Problem π ist PSPACE-vollständig, wenn es in PSPACE ist, $\pi \in$ PSPACE und jedes Problem aus PSPACE auf π in polynomieller Zeit reduzierbar ist, $\forall \pi' \in \mathrm{PSPACE} \ : \ ?\pi' \leq_p \pi$.
Problem: Quantified Boolean Formulas (QBF)
Gegeben: Eine Menge von Variablen $x_1, \ldots, x_n$ und einen Booleschen Ausdruck F in konjunktiver Normalform (KNF) über $x_1, \ldots, x_n$.
Frage: Ist die folgende quantifizierte Ausage wahr?

$$\exists x_1 \ \forall x_2 \ \exists x_3 \ \forall x_4 \ \cdots \ : \ F$$

Bemerkungen: QBF ohne $\forall$-Quantoren entspricht dem SAT Problem.

Satz: QBF $\in$ PSPACE
Beweis: Betrachte folgenden binären Baum B:



Jedes Blatt hat eine Markierung $w \in \{0,1\}^n$. Die Knoten auf Level i sind mit den Variablennamen x_i markiert.
Konstruiere aus B den folgenden Schaltkreis C:
Die Knoten auf Level $1, 3, 5, \ldots$ (Level $2, 4, 6, \ldots$) werden zu $\wedge$-Gattern ($\vee$-Gattern). Die Blätter werden die Inputknoten. Ein Blatt mit Markierung w (z.B. $w = 0110\ldots$) wird genau dann mit "1" markiert, wenn die Variablenbelegung entsprechend des Wortes w ($x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, \ldots$) eine erfüllende Belegung für F ist.
Der Schaltkreis berechnet die quantifizierte Aussage:

$$\exists\ x_1\ \forall\ x_2\ \exists\ x_3\ \forall\ x_4\ \ldots \quad :\ F$$

Der Schaltkreis kann (ohne daß er zuerst vollständig konstruiert werden muß !) auf Platz „Höhe von C+Größe von F" ausgewertet werden. $\square$
Satz: QBF ist PSPACE-schwer.
Folgerung: QBF ist PSPACE-vollständig.
Bemerkunken: Sei C eine Komplexitätsklasse. Eine Menge L (ein Problem π) ist C-vollständig, wenn $L \in$ C und jede Sprache $L' \in$ C (jedes Problem $\pi' \in$ C auf L (auf π) reduzierbar ist. Dies macht jedoch nur dann einen Sinn, wenn die Reduktion geringere Resourcen benötigt, als in der Klasse C erlaubt ist.
Beispiel:
NP-vollständig $\Rightarrow$ Reduktion deterministisch in polynomieller Zeit oder nichtdeterministisch auf logarithmischen Platz, oder deterministisch auf logarithmischen Platz
PSPCAE-vollständig $\Rightarrow$ Reduktion nichtdeterministisch in polynomieller Zeit, deterministisch in polynomieller Zeit, nichtdeterministisch auf polynomiellen Platz, oder derterministisch auf logarithmischen Platz.

Bemerkung: Selbst polynomiell platzbeschränkte Transformationen sind transitiv.
Satz: Seien $T_1 : \Sigma^* \to \Sigma^*$ und $T_2 : \Sigma^* \to \Sigma^*$ zwei $f(n)$ platzbeschränkte Transformationen. Dann gibt es eine $O(f(n))$ platzbeschränkte Transformation $T_3 : \Sigma^* \to \Sigma^*$ mit $T_3(x) = T_2(T_1(x))$.
Beweis: Seien M_1 und M_2 die $f(n)$-platzbeschränkten Turingmaschinen, die T_1 bzw. T_2 berechnen.
Eine Turingmaschine M_3 kann wie folgt konstruiert werden:

Input	x
	$\downarrow$
	M_1
	$\downarrow$
Output	$T_1(x)$
Input	$T_1(x)$
	$\downarrow$
	M_2
	$\downarrow$
Output	$T_2(T_1(x))$

M_3 arbeitet wie M_2. Wenn das i-te Symbol auf dem Eingabeband für M_2 gelesen werden soll, berechnet M_3 mit Eingabe x wie M_1 das i-te Symbol auf dem Ausgabeband. Danach setzt M_3 die Berechnung mit M_2 fort usw. □
Satz: QBF ist PSPACE-schwer.
Folgerung: QBF ist PSPACE-vollständig.

Bemerkung: Viele Spielprobleme sind PSPACE-vollständig.
Spielprobleme haben oft die folgende Form: Gibt es einen Zug für Weiß, so daß es für alle Züge von Schwarz einen Zug von Weiß gibt, ... so daß letztendlich Weiß gewonnen hat?
Problem: GENERALIZED HEX
Gegeben: Ein ungerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$.
Eine Position ist ein Tripel (V_1, V_2, V_3) mit $V_1, V_2, V_3 \subseteq V$ und $\{s, t\} \subseteq V_3$. Die Ausgangsposition ist $(\emptyset, \emptyset, V)$. Die Positionen $(V_1, V_2, \{s, t\})$ sind Endpositionen. Weiß (Schwarz) ist am Zug, wenn $\mid V_1 \mid + \mid V_2 \mid$ gerade (ungerade) ist. Weiß (Schwarz) bewegt einen Knoten aus $V_3 - \{s, t\}$ nach V_1 (V_2). Eine Endposition ist gewonnen für Weiß (Schwarz), wenn der von $V_1 \cup \{s, t\}$ ($V_2 \cup \{s, t\}$) induzierte Teilgraph von G einen Weg zwischen s und T enthält.
Frage: Gibt es eine erzwungene Gewinnstrategie für Weiß?
Satz: GENERALIZED HEX ist PSPACE-vollständig.

21 Übungsaufgaben

1. Betrachten Sie die folgenden acht Funktionen $f_i : \mathbb{N}_0 \to \mathbb{N}_0$, $i = \{1, \ldots, 8\}$. (Wie in der Vorlesung verstehen wir unter $f(n)$ immer $\max\{0, \lceil f(n) \rceil\}$.)

$$f_1(n) = n^2 \quad f_2(n) = n^3 \quad f_3(n) = n^2 \log(n)$$
$$f_4(n) = 2^n \quad f_5(n) = \log(n) \quad f_6(n) = \sqrt{n}$$
$$f_7(n) = 3^n \quad f_8(n) = \begin{cases} n, & n \text{ ungerade} \\ 2^n & \text{sonst} \end{cases}$$

Fertigen Sie nach folgenden Muster eine Tabelle an, in der Sie für jedes Paar (f, g) von Funktionen $f, g \in \{f_1, \ldots, f_8\}$ genau dan ein $\times$ in das Feld der Zeile f und der Spalte g setzen, wenn $f \in O(g)$ gilt.

$$f \in O(g)$$

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
f_1								
f_2								
f_3								
f_4								
f_5								
f_6								
f_7								
f_8								

Geben Sie zwei weitere Tabellen an, in denen die Fälle $f \in \Omega(g)$ und $f \in \Theta(g)$ betrachtet werden.

2. Ordnen Sie die folgenden 12 Funktionen aufsteigend nach ihrer asymptotischen Komplexität im O-Kalkül von schwach bis stark anwachsend. Kennzeichnen Sie Funktionen mit gleicher asymptotischer Komplexität.

$$n^2 \qquad \log(n) + \sqrt{n} \qquad \sqrt{n} \cdot n^{1.5} \qquad \sqrt{n} \qquad n^{1.5} \qquad 1.618^n$$

$$2^n - n^2 \qquad \log(n) \cdot \sqrt{n} \qquad \frac{\sqrt{n}}{n^{-1.5}} \qquad \frac{n}{\log(n)} \qquad n^1 \qquad \log(n)$$

3. Welche der folgenden Aussagen sind richtig bzw. falsch? Begründen Sie Ihre Antwort.

   (a) $O(\log(\log(n))) \subseteq O(\log^2(n))$

   (b) $O((\log(n))^2) = O(\log(n^2))$

   (c) $O((\sqrt{n})^4) = O(\sqrt{n^4})$

   (d) $O(n^{\frac{5}{2}}) = O(\frac{n^5}{n^2})$

   (e) $O(\log(\sqrt{n})) = O(\sqrt{\log(n)})$

   (f) $\forall\ f \in O(n) : O(f)$ ist echte Teilmenge von $O(f^2)$

   (g) $\exists\ f \in O(n^2) : \Omega(f(n)) \cap O(f(n)) = \emptyset$

   (h) $\Theta(f) = O(f) \cap \Omega(f)$

   (i) $\Omega(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists\ c, x_0 \in \mathbb{N}_0 : \forall x \in \mathbb{N}_0 : x \geq x_0 \Rightarrow g(x) \leq c \cdot f(x)\}$

   (j) $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$ und $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$

4. Beweisen oder widerlegen Sie die folgenden Behauptungen.

   (a) $\forall\ c \in \mathbb{N} : \forall\ f : \mathbb{N}_0 \to \mathbb{N}_0 : c \cdot f(n) \in O(f(n))$

   (b) $\forall\ f, g : \mathbb{N}_0 \to \mathbb{N}_0 : f + g \in O(\max\{f, g\})$

   (c) $\forall\ k \in \mathbb{N} : (\log\ n)^k \in O(n)$

   (d) $\forall\ c \in \mathbb{N}, c \geq 2 : \log_c(n) \in O(\log_2(n))$

(e) $\sum_{i=1}^{n} i \quad \in O(n)$

(f) $\forall\ c \in \mathbb{N} : c + \frac{1}{n} \in O(n)$

(g) $\forall\ k \in \mathbb{N} : n^k \in O(2^n)$

5. Geben Sie im O-Kalkül eine möglichst gute Abschätzung der Worst-Case-Laufzeit der folgenden drei Algorithmen an, die für zwei natürliche Zahlen $n, m \geq 1$ jeweils den Rest der Division von n durch m berechnen. Die Laufzeit soll dabei in Abhängigkeit von der Summe $n+m$ angegeben werden und nicht in Abhängigkeit von der Eingabegröße. (Die Eingabegröße wäre entweder 2, wenn wir nur die Anzahl der Zahlen zählen, bzw. $O(\log n + \log m)$, wenn wir die Größe der Binärdarstellung von n und m betrachten.

   Algorithmus: modulo
   Eingabe: $n, m \in \mathbb{N}_0$
   Ausgabe: Rest $\in \mathbb{N}_0$

   (a) Algorithmus 1:

   ```
   while (n ≥ m) n:= n - m;
   int Rest:=n;
   ```

   (b) Algorithmus 2:

   ```
   int Rest:= n - ( (n / m) * m);
   ```

   (c) Algorithmus 3:

   ```
   int x:= m;
   while (x ≤ n) x:= x * 2;
   while (x>m) {
        x:= x/2;
        if (n ≥ x) n:= n-x;
        }
   int Rest := n;
   ```

6. Sortieren Sie die nachfolgende Zahlenfolge mit

   (a) Sortieren durch Auswahl,

(b) Sortieren durch Einfügen,

(c) Bubblesort und

(d) Shakersort.

$$9, 5, 4, 2, 1, 8, 7, 6$$

Geben Sie nach jeder Runde die aktuelle Zahlenfolge an.

7. Sortieren Sie die nachfolgende Zahlenfolge mit Shellsort. Wählen Sie als Inkremente alle Zahlen der Form $2^p \cdot 3^q$, die kleiner als N sind, wobei $p, q \geq 0$ zwei ganze Zahlen sind.

$$6, 8, 2, 4, 10, 5, 9, 1, 3, 7$$

Geben Sie nach jeder Runde die aktuelle Zahlenfolge an.

8. Sortieren Sie die folgende Zahlenfolge mit Quicksort.

$$9, 5, 4, 2, 1, 8, 7, 6$$

Geben Sie vor jedem rekursiven Aufruf von Quicksort die betrachtete Teilfolge an. Wählen Sie als Pivotelement

(a) das letzte Element der Folge,

(b) die 3-Median-Strategie,

(c) das kleinste Element der Folge bzw.

(d) den Median m der Folge, d.h. in einer Folge mit N Zahlen gibt es $\lceil \frac{N-1}{2} \rceil$ Zahlen die kleiner gleich m sind und $\lfloor \frac{N-1}{2} \rfloor$ Zahlen die grösser gleich m sind.

9. Geben Sie die Laufzeiten (in O-Notation) der Sortierverfahren „Sortieren durch Auswahl“, „Sortieren durch Einfügen“ und „Bubblesort“ im schlimmsten Fall an, wenn nur Eingaben aus den Mengen I_1, I_2, I_3 bzw. I_4 betrachtet werden.

$$I_1 = \{(1, \tfrac{N}{2}+1, 2, \tfrac{N}{2}+2, \ldots, \tfrac{N}{2}) \mid N \in \mathbb{N}, N \text{ gerade}\}$$
$$I_2 = \{(N, 1, N-1, 2, N-2, 3, \ldots, N-\tfrac{N}{2}+1, \tfrac{N}{2}) \mid N \in \mathbb{N}, N \text{ gerade}\}$$
$$I_3 = \{(N, 1, 2, 3, \ldots, N-1) \mid N \in \mathbb{N}\}$$
$$I_4 = \{(2, 3, 4, \ldots, N, 1) \mid N \in \mathbb{N}\}$$

10. Geben Sie größenordnungsmäßig im O-Kalkül die Komplexität der folgenden Operationen auf einem Heap mit N Elementen im schlimmsten Fall an. Am Ende einer jeden Operation soll stets ein Heap zurückbleiben.

    (a) Suchen eines beliebigen Elementes,

    (b) Entfernen eines beliebigen Elementes,

    (c) Suchen des Elementes mit minimalen Schlüssel und

    (d) Entfernen des Elementes mit minimalen Schlüssel.

11. Gegeben sei die Schlüsselfolge

$$13, 6, 1, 15, 3, 2, 8, 4, 5, 7, 11, 12, 10, 14, 9$$

    Konstruieren Sie aus der obigen Schlüsselfolge einen Heap in einem Feld a mit 15 Elementen. Sortieren Sie die Folge in aufsteigender Reihenfolge mit dem Verfahren „Heapsort“ und geben Sie alle Zwischenschritte an.

$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$	$a[10]$	$a[11]$	$a[12]$	$a[13]$	$a[14]$	$a[15]$
13	6	1	15	3	2	8	4	5	7	11	12	10	14	9
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

12. Überprüfen und erklären Sie, ob es für die folgenden Sortierverfahren einfache Implementationen gibt, die stabil sind (d.h. die Reihenfolge von Elementen mit gleichem Schlüssel wird während des Sortierens nicht verändert):

    (a) Sortieren durch Auswahl,

    (b) Sortieren durch Einfügen,

(c) Shellsort,

(d) Bubblesort und

(e) Quicksort.

13. Entwerfen Sie einen Algorithmus der zu einer gegebenen Objektfolge einen Heap von links nach rechts aufbaut. Ihr Algorithmus soll eine Methode verwenden, die einen gegebenen Heap um ein Objekt erweitert.

    Geben Sie die Laufzeit für den Aufbau eines Heaps mit Ihrem Algorithmus an.

14. Modifizieren Sie den Algorithmus „Heapsort“, so dass ein Element drei Nachfolger anstelle von zwei Nachfolgern besitzt. Erläutern Sie, wie ein solcher Heap in einem Feld (Array) dargestellt werden kann. Geben Sie Algorithmen für das Erstellen eines Heaps, das Versickern eines Elements und das Sortieren einer Zahlenfolge an.

    Geben Sie die maximale Anzahl der benötigten Vergleiche beim initialen Heapaufbau, beim Versickern und beim Sortieren mit obigem Heapsort an.

15. Sortieren Sie die Schlüsselfolge

    $$6, 12, 17, 4, 3, 21, 9, 5, 10, 11, 2, 1, 13, 8, 22, 7$$

    mit den Sortierverfahren

    (a) 2-Wege-Mergesort,

    (b) reines 2-Wege-Mergesort und

    (c) natürliches 2-Wege-Mergesort.

16. Sortieren Sie die angegebene Zahlenfolge durch Fachverteilung. Geben Sie dabei die Belegung der einzelnen Fächer nach jeder Verteilphase an und jeweils die Folge, die nach einer Sammelphase enstanden ist.

    1235, 2479, 7421, 4128, 5411, 4009, 6088, 9949, 7899, 6123,
    3110, 4142, 7600, 0318, 8632, 3038, 5259, 4300, 8748, 6200

17. Wenn Sie die folgenden Zahlen zu sortieren hätten und dabei entweder Sortieren durch Fachverteilung oder ein Verfahren, das nur mit Schlüsselvergleichen arbeitet, verwenden könnten, welche Überlegungen würden Sie anstellen?

    $$k_1 = 12345678,\ k_2 = 43482809,\ k_3 = 91929390,\ \ldots,k_{10} = 91929397$$

Mit welchen der aus der Vorlesung bekannten Sortierverfahren würden Sie die obige Folge sortieren?

18. Entwerfen Sie einen Algorithmus für Radixsort, der mit Hilfe von Arrays wie in der Vorlesung vorgestellt arbeitet. Die Laufzeit Ihrer Löung soll in $O(b \cdot (N + m))$ und der benötigte Speicherplatz soll in $O(N + m)$ liegen.

    Hinweis: Bestimmen Sie zu Beginn eines Durchlaufs wieviele Elemente auf jedes Fach fallen.

19. Geben Sie jeweils (falls möglich) eine Folge F mit sechs Schlüsseln an, für die

    (a) $\text{runs}(F) < \text{inv}(F)$

    (b) $\text{runs}(F) < \text{rem}(F)$

    (c) $\text{inv}(F) < \text{runs}(F)$

    (d) $\text{inv}(F) < \text{rem}(F)$

    (e) $\text{rem}(F) < \text{inv}(F)$

    (f) $\text{rem}(F) < \text{runs}(F)$

    gilt.

20. Entwerfen Sie Algorithmen, die die Inversionszahl in $O(n^2)$, die Run-Zahl in $O(n)$ und die LAS-Zahl in $O(n^2)$ bestimmen.

21. Als weiteres Vorsortierungsmaß findet man in der Literatur $\text{exc}(F)$, das ist die kleinste Anzahl von Vertauschungen, die man benötigt, um eine Folge F in aufsteigende Ordnung zu bringen.

    (a) Geben Sie jeweils eine Folge F mit N Schlüsseln an, für die

        i. $\text{exc}(F)$ maximal ist,

        ii. $\text{exc}(F) = \lfloor \frac{N}{2} \rfloor$.

    (b) Welche Beziehung gilt zwischen $\text{exc}(F)$ und $\text{inv}(F)$ für alle Folgen F?

22. Analysieren Sie die Median-der-Mediane Strategie, wobei die N Elemente der Folge in

    (1) $\lfloor \frac{N}{3} \rfloor$ Gruppen mit 3 Elementen und höchstens eine Gruppe mit weniger als 3 Elementen, bzw.

    (2) $\lfloor \frac{N}{7} \rfloor$ Gruppen mit 7 Elementen und höchstens eine Gruppe mit weniger als 7 Elementen

    aufgeteilt werden.

(a) Geben Sie die Rekursionsgleichung an, die die Anzahl der Schritte abschätzt, die erforderlich ist, um das Element mit i-t kleinstem Schlüssel zu finden.

(b) Bestimmen Sie wie in der Vorlesung ein c in Abhängigleit von a und das dazugehörige minimale N, für dass die Suche theoretisch in linearer Zeit erfolgt.

(c) Zeigen Sie die lineare Laufzeit des Verfahrens für den Fall (2).

(d) Begründen Sie, warum im Fall (1) mit der Beweisidee aus der Vorlesung keine lineare Laufzeit des Verfahrens gezeigt werden kann.

23. (a) Führen Sie in dem Feld mit den folgenden Schlüsseln

$$1, 3, 7, 9, 12, 13, 21, 22, 28, 30, 38, 42, 56, 60, 61$$

eine Binärsuche nach dem Element mit Schlüssel 38 durch.

(b) Führen Sie in dem Feld mit den folgenden Schlüsseln

$$1, 3, 5, 10, 11, 15, 21, 22, 28, 30, 38, 42, 55, 60, 61, 64, 69, 70, 71, 78$$

eine Fibonaccisuche nach dem Element mit Schlüssel 38 durch.

(c) Führen Sie in dem Feld mit den folgenden Schlüsseln

$$1, 5, 7, 10, 12, 15, 19, 22, 25, 30, 39, 42, 54, 60, 67$$

eine Exponentielle Suche nach dem Element mit Schlüssel 19 durch.

(d) Führen Sie in dem Feld mit den folgenden Schlüsseln

$$1, 3, 9, 10, 12, 18, 21, 25, 28, 30, 39, 44, 58, 60, 66$$

eine Interpolationsuche nach dem Element mit Schlüssel 18 durch.

Geben Sie dabei an, welche Schlüssel miteinander verglichen werden.

24. Geben Sie für jedes Paar V_1, V_2 von Suchverfahren (sequentielle Suche, binäre Suche, Fibonacci-Suche, exponentielle Suche, Interpolationssuche) einen Suchschlüssel k und zwei Zahlenfolgen A_1, A_2 an, so dass in A_1 die Suche nach k mit V_1 weniger Schlüsselvergleiche benötigt als mit V_2, und in A_2 die Suche nach k mit V_2 weniger Schlüsselvergleiche benötigt als mit V_1, falls dies überhaupt möglich ist.

25. Gegeben sei eine sortierte Liste mit 20 Elementen, die in einem Feld der Länge 20 abgespeichert sind. Geben Sie für jeden beliebigen Suchschlüssel k an, in welcher Reihenfolge die Schlüssel der Listenelemente mit k verglichen

werden, wenn die Fibonacci-Suche als Suchverfahren verwendet wird. Stellen Sie dazu den entsprechenden Suchbaum für Listen der Länge 20 dar. Die Knoten des Suchbaumes sollen mit den Feldpositionen markiert werden, mit denen Suchschlüssel k verglichen wird.

Berechnen Sie die im Mittel erforderliche Anzahl von Schlüsselvergleichen beim Durchsuchen einer Liste mit 20 Elementen mit Fibonacci-Suche, wobei vorausgesetzt wird, dass die Zugriffswahrscheinlichkeit für alle Elemente gleich gross ist und nur nach Elementen gesucht wird die auch in der Liste gespeichert sind.

26. Gegeben sei die Liste $L = (1, 2, 3, 4, 5, 6, 7)$ und die Zugriffsfolge s mit 21 Zugriffen:

$$7, 2, 7, 3, 3, 7, 4, 4, 4, 7, 5, 5, 5, 5, 7, 6, 6, 6, 6, 6, 7$$

Vergleichen Sie das Verhalten der MF- und der T-Regel für die Zugriffsfolge s, indem Sie das folgende Schema ergänzen.

nächstes Element von s	L_{MF}	Zugriffskosten MF-Regel	L_{T}	Zugriffskosten T-Regel
–	$1, 2, 3, 4, 5, 6, 7$	–	$1, 2, 3, 4, 5, 6, 7$	–
7	$7, 1, 2, 3, 4, 5, 6$	7	$1, 2, 3, 4, 5, 7, 6$	7
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	Gesamtkosten:	.....	Gesamtkosten:	.....

27. Gegeben sei das Muster `abrakadabra` der Länge 11. Berechnen Sie für dieses Muster die Werte $next[j]$ für alle j mit $0 \leq j \leq 10$. Suchen Sie das Muster im folgenden Text mit dem Verfahren von Knuth-Morris-Pratt.

```
und abraham sprach abrakadabra, aber ...
```

28. Geben Sie die Belegung einer Hash-Tabelle der Größe 13 an, wenn die Schlüssel

$$5, 1, 19, 23, 14, 17, 32, 30, 2$$

in eine anfangs leere Tabelle eingefügt werden und offenes Hashing mit Hash-Funktion $h(k) = k \bmod 13$ sowie

(a) linearem Sondieren,

(b) quadratischem Sondieren, bzw.

(c) Sondieren mit Sondierungsfunktion $s(j,k) = j \cdot ((k \mod 11)+1)$ (double Hashing)

verwendet wird.

Vergleichen Sie die Anzahlen der beim Einfügen betrachteten Hashtabellenplätze für diese drei Sondierungsverfahren.

Wie groß sind die Anzahl der Vergleiche insgesamt, wenn in der gefüllten Hash-Tabelle nach jedem vorhandenen Schlüssel einmal gesucht wird?

29. Wieviele Schritte werden im schlechtesten Fall bzw. im Mittel benötigt, um n Schlüssel in eine anfangs leere Hash-Tabelle der Größe M einzufügen, wenn zur Überlaufbehandlung die Methode der Verkettung mit unsortierten bzw. sortierten Listen verwendet wird? Wieviele Schritte benötigt man in diesen Fällen, um in der gefüllten Hash-Tabelle nach jedem der n eingefügten Schlüssel einmal zu suchen?

30. Untersuchen Sie, ob die folgende Menge von Hashfunktionen

$$\{h_{i,j} : \{0,\ldots,9\} \to \{0,\ldots,9\} : h_{i,j}(k) = (k^i + j) \mod 10 \mid i = 1,\ldots 10;\ \ j = 0,\ldots,9\}$$

eine universelle Menge von Hashfunktionen ist. Begründen Sie Ihre Antwort.

31. Geben Sie die Belegung einer Hash-Tabelle der Größe 7 an, wenn die Schlüssel

$$11, 3, 10, 45, 38, 5$$

mit Brents Algorithmus in eine anfangs leere Tabelle eingefügt werden.

Verwenden Sie $h(k) = k \mod 7$ und als Sondierungsfunktion

$$s(j,k) = j \cdot h'(k) \text{ mit } h'(k) = 1 + k \mod 5.$$

32. Gegeben sei die Folge F von acht Schlüsseln

$$F = 4, 8, 7, 2, 5, 3, 1, 6.$$

(a) Geben Sie den zu F gehörenden Suchbaum an, der entsteht, wenn die Schlüssel der Reihe nach in den anfangs leeren Baum eingefügt werden.

(b) Geben Sie die Knoten des in (a) erzeugten Baumes in der Hauptreihenfolge, Nebenreihenfolge und in der symmetrischen Reihenfolge an.

(c) Welcher Baum entsteht aus dem in (a) erzeugten Baum, wenn man den Schlüssel 4 entfernt?

33. Gegeben sei ein Suchbaum B mit ganzzahligen Schlüsseln. Gegeben sei außerdem ein Schlüssel x. Gesucht ist der größte Schlüssel in B, der kleiner oder gleich x ist.

    Geben Sie einen Algorithmus an, der diese Aufgabe in $O(h)$ Schritten löst, wenn h die Höhe von B ist.

34. (a) Geben Sie alle Suchbäume mit vier inneren Knoten an, die jeweils von genau einer Permutation der Zahlen $1, \ldots, 4$ erzeugt werden, und nennen Sie die Permutationen.

    (b) Geben Sie einen Suchbaum mit zehn inneren Knoten an, der von genau zwei Permutationen der Zahlen $1, \ldots, 10$ erzeugt wird, und nennen Sie die Permutationen.

35. Geben Sie den AVL-Baum an, der durch Einfügen der Schlüssel

$$10, 15, 11, 4, 8, 7, 3, 2, 13$$

    in den anfangs leeren Baum entsteht.

36. (a) Geben Sie den AVL-Baum an, der durch Einfügen der Schlüssel

$$14, 15, 17, 12, 11, 13, 16$$

    in den anfangs leeren Baum entsteht. Geben Sie an welche Rotationen durchgeführt werden und zeichnen Sie die dabei entstehenden Bäume.

    (b) Geben Sie den AVL-Baum an, der durch Löschen des Schlüssels 14 in dem Baum aus (1.) entsteht

37. (a) Geben Sie den Splay-Baum an, der durch Einfügen der Schlüssel

$$1, 3, 5, 7, 9, 11, 13, 15, 2, 6$$

    in den anfangs leeren Baum entsteht. Zeichnen Sie die dabei entstehenden Bäume.

    (b) Entfernen Sie nacheinander die Schlüssel

$$13, 11, 7, 6, 3.$$

38. (a) Fügen Sie die Schlüssel

$$2, 6, 1, 3, 24, 7, 8, 13, 14, 26, 25, 15, 17, 19, 21$$

    in einen anfangs leeren B-Baum der Ordnung 3 ein. Zeichnen Sie die dabei entstehenden Bäume.

(b) Entfernen Sie nacheinander die Schlüssel

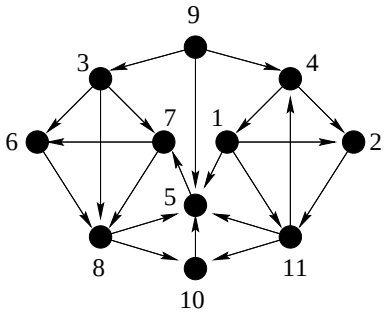
$$21, 19, 17, 15, 25$$

aus dem B-Baum der Ordnung 3 aus (a). Zeichnen Sie die dabei entstehenden Bäume.

39. Geben Sie für den folgenden Graphen eine Darstellung als

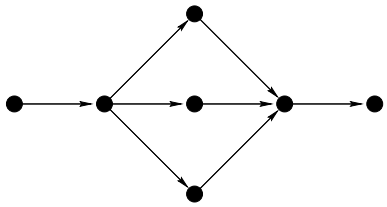
- Adjazenzmatrix
- Adjazenzlisten

an.



Was ändert sich wenn man den zugehörigen ungerichteten Graphen betrachtet?

40. Berechnen Sie nach dem in der Vorlesung vorgestellten Algorithmus eine topologische Sortierung des folgenden Graphen. Wieviele verschiedene topologische Sortierungen gibt es in diesem Beispiel?



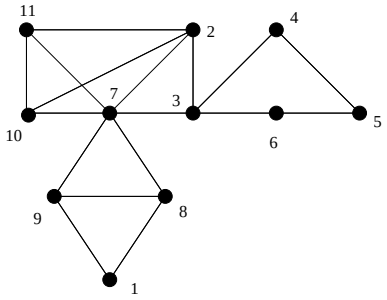
41. Geben Sie einen Graphen mit 6 Knoten und 8 Kanten an, so dass die Anzahl der verschiedenen topologischen Sortierungen möglichst groß ist.

42. Bestimmen Sie für den Graphen aus Aufgabe 40 den transitiver Abschluss.

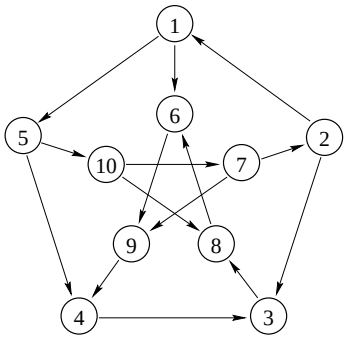
43. Betrachten Sie den Graphen aus Aufgabe 39. Starten Sie bei Knoten 9 mit einer Tiefensuche. Geben Sie die DFB- und DFE-Numerierung an, und geben Sie bei jeder Kante an, ob es sich um eine Baum-, Vorwärts-, Rückwärts- oder Seitwärtskante handelt. Im Falle einer Wahlmöglichkeit sollen zuerst die Knoten mit kleineren Knotennummern betrachtet werden.

44. Betrachten Sie den Graphen aus Aufgabe 39. Bestimmen Sie die Reihenfolge der Knoten, in der sie bei einer Breitensuche als "besucht" markiert werden, starten Sie bei Knoten 9. Im Falle einer Wahlmöglichkeit sollen zuerst die Knoten mit kleineren Knotennummern betrachtet werden.

45. Geben Sie für den folgenden Graphen die Artikulationspunkte, die zweifach zusammenhängenden Komponenten (mit mindestens drei Knoten) und die dreifach zusammenhängenden Komponenten (mit mindestens vier Knoten) an.



46. Bestimmen Sie für den folgenden Graphen die Knotenmengen der starken Zusammenhangskomponenten.



47. (a) Geben Sie den Linksbaum an, der durch Einfügen der Schlüssel

$$28, 6, 8, 14, 3, 30, 23$$

in den anfangs leeren Baum entsteht.

(b) Entfernen Sie den Schlüssel 28.

48. (a) Geben Sie den Binomialbaum an, der durch Einfügen der Schlüssel

$$22, 25, 8, 18, 6, 15, 13, 9, 7$$

in den anfangs leeren Baum entsteht. Zeichnen Sie die dabei entstehenden Bäume.

(b) Entfernen Sie den Schlüssel 8.

49. (a) Bauen Sie einen neuen Fibonacci-Heap mit den Schlüsseln

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

auf, und entfernen Sie den Minimalknoten 1 aus der Wurzelliste.

(b) Erniedrigen Sie Schlüssel 8 auf 1.

(c) Entfernen Sie Schlüssel 5.

50. Beweisen oder wiederlegen Sie, dass die Höhe der Bäume in einem Fibonacci-Heap mit n Knoten höchstens $\log_2(n)$ beträgt.

51. Eine Folge von n Operationen wird auf einer Datenstruktur ausgeführt, wobei die i-te Operation Kosten $1 + i$ verursacht, wenn i eine Zweierpotenz ist. Ansonsten kostet die i-te Operation eine Einheit.

Bestimmen Sie die amortisierten Kosten pro Operation mit Hilfe des Bankkonto-Paradigmas. Geben Sie eine Kontostandsfunktion Φ an, so dass die amortisierten Kosten a_i, $1 \leq i \leq M$, der i-ten Operation aus $O(1)$ sind.

Füllen Sie nachfolgende Tabelle mit den amortisierten Kosten, tatsächlichen Kosten und Kontoständen für $i = 1, \ldots, 9$ auf.

i	t_i	Φ_i	a_i
1			
2			
3			
4			
5			
6			
7			
8			
9			

52. Betrachten Sie die nachfolgende Implementierung der Datenstruktur Stack. Die Elemente im Stack werden der Reihe nach in einem Array A abgelegt. Variable x speichert die Anzahl der eingetragenen Elemente.

Die Operation push(S, e) überprüft zuerst, ob das Array noch Elemente aufnehmen kann. Ist $A.length$ größer als x, dann wird das Element e am Ende

des Arrays A eingefügt. Ansonsten wird ein neues Array definiert, das doppelt so viele Elemente speichern kann, wie A. Die Elemente aus dem alten Array werden in das neue Array kopiert und das Element e wird eingetragen.

Die Operation pop(S) entfernt das letzte Element in Array, indem die Variable x um 1 erniedrigt wird. Ist anschließend x kleiner oder gleich $A.length/4$, so wird ein neues Array definiert, das nur halb so viele Elemente speichern kann wie A. Die Elemente aus dem alten Array werden in das neue Array kopiert.

Analysieren Sie die Kosten für eine Folge s von M Operationen (N push() Operationen und $M - N$ pop() Operationen) mit Hilfe des Bankkonto-Paradigmas. Geben Sie eine Kontostandsfunktion Φ an, so dass die amortisierten Kosten a_i, $1 \leq i \leq M$, der i-ten Operation aus $O(1)$ sind. Die tatsächlich Kosten einer push() oder pop() Operationen sind 1 bzw. $1 + x$, falls umkopiert werden muss.

Füllen Sie nachfolgende Tabelle mit den amortisierten Kosten, tatsächlichen Kosten und Kontoständen für die angegebene Folge von Operationen.

	t_i	Φ_i	a_i
$\text{push}(S, e_1)$			
$\text{push}(S, e_2)$			
$\text{push}(S, e_3)$			
$\text{push}(S, e_4)$			
$\text{push}(S, e_5)$			
$\text{push}(S, e_6)$			
$\text{push}(S, e_7)$			
$\text{push}(S, e_8)$			
$\text{push}(S, e_9)$			
$\text{push}(S, e_{10})$			
$\text{push}(S, e_{11})$			
$\text{push}(S, e_{12})$			
$\text{push}(S, e_{13})$			
$\text{push}(S, e_{14})$			
$\text{push}(S, e_{15})$			
$\text{push}(S, e_{16})$			
$\text{push}(S, e_{17})$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			
$\text{pop}(S)$			

```
public class stack {
private int [] A;
private int x;

stack ()
{
  A = new int [1];
  x = 0;
}

public boolean empty ()
{
  return x == 0;
}

public void push (int e)
{
  if (x >= A.length) {
    int [] B = new int [A.length*2];
    for (int i = 0; i < x; i++)
      B[i] = A[i];
    A = B; }
  A[x++] = e;
}

public void pop ()
{
  if (x > 0) {
     x--;
     if ((x>1)&&(x <= A.length/4)) {
        int [] B = new int [A.length/2];
        for (int i = 0; i < x; i++)
          B[i] = A[i];
        A = B; }
     }
}

public int top ()
{
  if (x > 0)
    return A[x-1];
  else
    return -1;
}

}
```

53. Sei $G = (V, E)$ ein Graph und $f : E \to \mathbb{R}$ eine Kantenbewertung. In der Vorlesung wurde die Länge eines Weges $p = (u_1, \ldots, u_k)$ als

$$L(p) := f((u_1, u_2)) + f((u_2, u_3)) + \cdots + f((u_{k-1}, u_k))$$

definiert. Betrachten Sie nun die folgenden 5 verschiedenen Definitionen für die Länge eines Weges $p = (u_1, \ldots, u_k)$ mit $k \geq 1$.

(a)

$$L_1(p) := \begin{cases} \min\{f((u_1, u_2)), f((u_2, u_3)), \quad \ldots, f((u_{k-1}, u_k))\} & \text{falls } k \geq 2 \\ 0 & \text{sonst} \end{cases}$$

(b)

$$L_2(p) := \begin{cases} \prod_{i=1}^{k-1} f((u_i, u_{i+1})) & \text{falls } k \geq 2 \\ 1 & \text{sonst} \end{cases}$$

(c)

$$L_3(p) := \begin{cases} \frac{L_3((u_1,\ldots,u_{k-1}))+f((u_{k-1},u_k))}{2} & \text{falls } k \geq 3 \\ f((u_{k-1}, u_k)) & \text{falls } k = 2 \\ 0 & \text{sonst} \end{cases}$$

(d)

$$L_4(p) := \begin{cases} L_4((u_1, \ldots, u_{k-1}))^2 + f((u_{k-1}, u_k))^2 & \text{falls } k \geq 3 \\ f((u_{k-1}, u_k))^2 & \text{falls } k = 2 \\ 0 & \text{sonst} \end{cases}$$

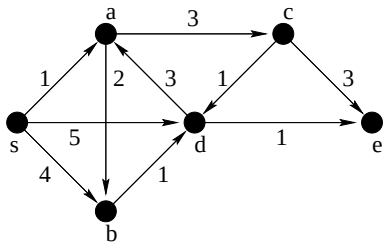
(e)

$$L_5(p) := \begin{cases} \frac{\sum_{i=1}^{k-1} f((u_i,u_{i+1}))}{k-1} & \text{falls } k \geq 2 \\ 0 & \text{sonst} \end{cases}$$

Untersuchen Sie, in wie weit die folgenden beiden Eigenschaften für die 5 Längenmaße $L_1, \ldots, L_5$ stimmen.

(a) Für jeden kürzesten Weg $p = (v_1, \ldots, v_k)$ von v_1 nach v_k ist jeder Teilweg $p' = (v_i, v_{i+1}, \ldots, v_j)$ mit $1 \leq i \leq j \leq k$ ein kürzester Weg von v_i nach v_j.

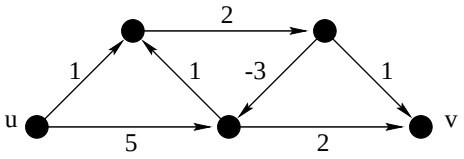
(b) Der kürzeste Wegewert $kv(u, v)$ ist für jedes Knotenpaar $u, v \in V$ definiert.



54. (a) Wenden Sie den in der Vorlesung vorgestellten „einfachen Kürzeste-Wege Algorithmus“, der die Dreiecksungleichungen wiederholt für alle Kanten in beliebiger Reihenfolge testet, auf den obigen Graphen an. Der Startknoten sei s.

(b) Betrachten Sie Dijkstras Algorithmus zur Bestimmung aller kürzester Wege von einem Startknoten s. Geben Sie nach jedem Schritt die Distanzwerte und die Elemente der Datenstruktur R an, die Dijkstras Algorithmus bei obigem Graphen berechnet. Der Startknoten sei s.

(c) Die längste Weglänge von einem Knoten u zu einem Knoten v ist die maximale Länge aller Wege von u nach v, falls mindestens ein solcher Weg existiert. Beschreiben Sie, wie man mit Hilfe von Kürzeste-Wege Algorithmen das Längste-Wegeproblem lösen kann. Veranschaulichen Sie Ihre Idee an folgendem Beispiel.

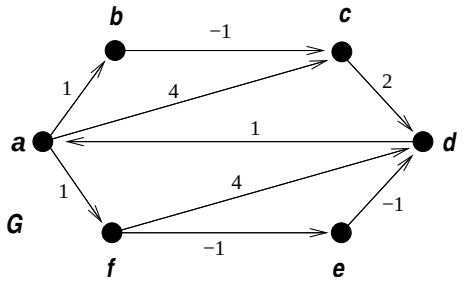


55. (a) Berechnen Sie den transitiven Abschluss von G.

(b) Bestimmen Sie mit dem Algorithmus von Floyd Warshall für jedes Knotenpaar u, v die kürzeste Weglänge zwischen u und v in G.

(c) Bestimmen Sie eine Kürzeste Wege Funktion für den Knoten a in G und transformieren Sie den Graphen G mit dieser Distanzfunktion in einen Distanzgraphen G'.

(d) Bestimmen Sie eine Kürzeste-Wege-Funktion für den Knoten d im Graphen G'.



56. Betrachten Sie folgenden Graphen (Abbildung 21.7). Geben Sie nach jedem Schritt der Algorithmen von Prim und Kruskal die gefärbten Kanten an. Starten Sie Prims Algorithmus bei Knoten a.

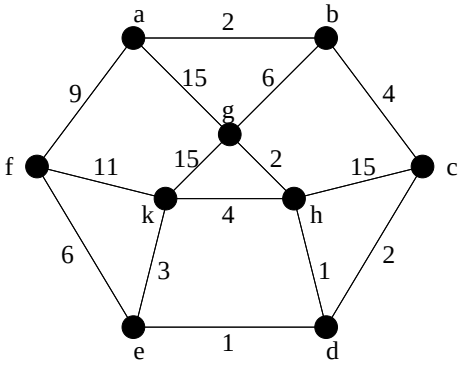


Abbildung 21.7: Graph für Aufgabe 56

57. Es sei $G = (V, E)$ ein ungerichteter Graph mit Kantengewichtsfunktion $f : E \to \mathbb{R}$. Entwerfen Sie einen Algorithmus, der für G einen minimalen Spannwald berechnet, der aus genau zwei Bäumen besteht. Schätzen Sie die Laufzeit Ihrer Lösung ab.

58. Es sei $G = (V, E)$ ein ungerichteter Graph mit Kantengewichtsfunktion $f : E \to \mathbb{R}$. Zeigen Sie, dass man einen minimalen Spannbaum in Linearzeit berechnen kann, wenn es in G höchstens 7 verschiedene Kantengewichte gibt.

59. Sei $G = (V, E)$ ein ungerichteter Graph mit einer Kostenfunktion $f : E \mapsto \mathbb{R}$. Seien T und T' zwei verschiedene minimale Spannbäume von G, und seien L sowie L' zwei aufsteigend sortierte Folgen der Kostenwerte der Kanten in T bzw. T'. Zeigen Sie, dass L und L' identisch sind.

60. Bestimmen Sie für das Netzwerk $G = (V, E)$ in Abbildung 21.8 mit Kapazitätsfunktion $c : E \mapsto \mathbb{R}_{\geq 0}$, Quelle $s \in V$ und Senke $t \in V$ eine Flussfunktion $f_{s,t}$ mit maximalem Fluss mit

(a) Karp/Edmonds Flussalgorithmus (Pfade mit maximaler Flussvergrösserung)

(b) Karp/Edmonds Flussalgorithmus (Pfade mit minimaler Anzahl von Kanten)

(c) Dinics Flussalgorithmus

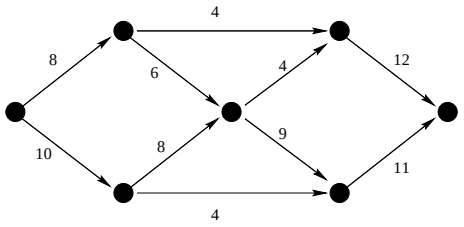


Abbildung 21.8: Netzwerk für Aufgabe 60

61. Gegeben Sei ein Netzwerk $G = (V, E)$ mit Kapazitätsfunktion $c : E \to \mathbb{R}_{\geq 0}$, einer Quelle $s \in V$ und einer Senke $t \in V$. Es sei $S \subseteq V$, $\overline{S} = V - S$ mit $s \in S$ und $t \in \overline{S}$ ein s,t-Schnitt und $f_{s,t}$ eine Flussfunktion für G bezüglich s und t.

Zeigen Sie die folgenden Aussagen:

(a)
$$F(f_{s,t}) = \sum_{e \in (S,\overline{S})} f_{s,t}(e) - \sum_{e \in (\overline{S},S)} f_{s,t}(e)$$

(b)
$$F(f_{s,t}) \leq \sum_{e \in (S,\overline{S})} c(e)$$

(c) Es gibt immer mindestens einen s,t-Schnitt $(S, \overline{S})$ und mindestens eine Flussfunktion $f_{s,t}$ für G bezüglich s und t mit
$$F(f_{s,t}) = \sum_{e \in (S,\overline{S})} c(e)$$

62. (a) Bestimmen Sie für den folgenden Graphen (Abbildung 21.9) ein maximales Matching nach der Methode der zunehmenden Pfade; ignorieren Sie die Bewertung der Kanten.

(b) Bestimmen Sie für den folgenden Graphen ein maximales gewichtetes Matching.

(c) Bestimmen Sie für den folgenden Graphen ein minimales, nicht erweiterbares Matching (d.h. es kann keine Kante zu dem Matching hinzugenommen werden, so dass man ein grösseres Matching erhält); ignorieren Sie die Bewertung der Kanten.

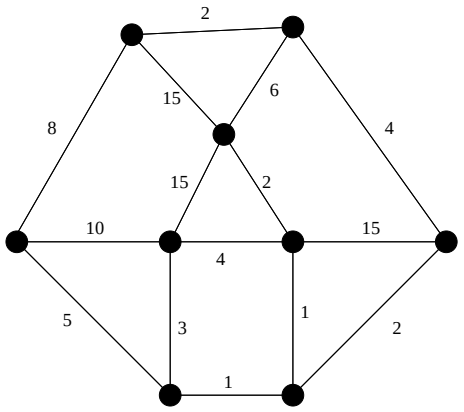


Abbildung 21.9: Graph für Aufgabe 62

63. Bestimmen Sie für das Netzwerk $G = (V, E)$ in Abbildung 21.10 mit Kapazitätsfunktion $c : E \mapsto \mathbb{R}_{\geq 0}$, Quelle $s \in V$ und Senke $t \in V$ eine Flussfunktion $f_{s,t}$ mit maximalem Fluss mit

    (a) Karp/Edmonds Flussalgorithmus (Pfade mit maximaler Flussvergrösserung)

    (b) Karp/Edmonds Flussalgorithmus (Pfade mit minimaler Anzahl von Kanten)

    (c) Dinics Flussalgorithmus

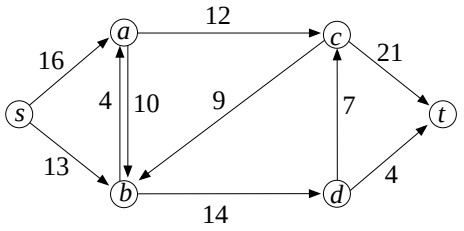


Abbildung 21.10: Netzwerk für Aufgabe 63

64. Zeigen Sie die folgenden Aussagen:

    (a) $T(n) = 2 \cdot T(\lceil \frac{n}{2} \rceil) + 7 \in \Theta(n)$

    (b) $T(n) = 2 \cdot T(\lceil \frac{n}{2} \rceil + 17) + 7 \in \Theta(n)$

    (c) $T(n) = 2 \cdot T(\lceil \frac{n}{2} \rceil) + n + 7 \in \Theta(n \cdot \log(n))$

    (d) $T(n) = 2 \cdot T(\lceil \frac{n}{2} \rceil) + 5 \cdot n + 7 \in \Theta(n \cdot \log(n))$

    (e) $T(n) = 2 \cdot T(\lceil \frac{n}{2} \rceil + 17) + 5 \cdot n + 7 \in \Theta(n \cdot \log(n))$

    (f) $T(n) = 2 \cdot T(\lceil \frac{n}{3} \rceil + 17) + 5 \cdot n + 7 \in \Theta(n)$

    (g) $T(n) = T(\lceil \sqrt{n} \rceil) + 1 \in \Theta(\log(\log(n)))$

65. Berechnen Sie mit dem in der Vorlesung vorgestellten Algorithmus eine minimale Triangulierung für das konvexe Polygon gegeben durch die Eckpunkte
$A_1 = (1, 2), A_2 = (1, 3), A_3 = (1, 4), A_4 = (2, 5),$
$A_5 = (3, 4), A_6 = (3, 3), A_7 = (3, 2), A_8 = (2, 1).$

    Geben sie alle Zwischenergebnisse $c_{i,j}$ mit $1 \leq i < j \leq 8$ an.

66. Betrachten Sie folgende 6×6-Kostenmatrix:

$$(c_{i,j}) = \begin{pmatrix} 1 & 2 & 8 & 4 & 7 & 6 \\ 2 & 3 & 5 & 1 & 6 & 9 \\ 8 & 5 & 7 & 3 & 4 & 1 \\ 4 & 1 & 3 & 1 & 5 & 3 \\ 7 & 6 & 4 & 5 & 6 & 2 \\ 6 & 9 & 1 & 3 & 2 & 4 \end{pmatrix}$$

    Eine Lösung für obige Matrix sei eine Permutation $\pi : \{1, \ldots, 6\} \rightarrow \{1, \ldots, 6\}$ mit Kosten

$$C(\pi) = c_{\pi(6),\pi(1)} + \sum_{i=1}^{5} c_{\pi(i),\pi(i+1)}.$$

    Eine benachbarte Lösung für eine gegebene Lösung π erhält man durch Vertauschen von zwei Werten $\pi(i)$ und $\pi(j)$ mit $1 \leq i, j \leq 6$.

    Berechnen sie mit der Greedy-Strategie eine nicht weiter verbesserbare Lösung. Geben Sie alle Zwischenlösungen an.

67. Betrachten Sie das folgende **Rucksackproblem**. Gegeben sei ein Rucksack mit einer Kapazität R und Objekte $O = \{o_1, \ldots, o_n\}$ denen ein Gewicht $w(o_i)$ und eine Größe $g(o_i)$, $1 \leq i \leq n$, zugeordnet ist. Eine Teilmenge $O' \subseteq O$ ist eine Lösung für das Rucksackproblem, falls $\sum_{o \in O'} g(o) \leq R$. Die Kosten einer Lösung $O' \subseteq O$ betragen $C(O') = \sum_{o \in O'} w(o)$. Gesucht ist eine Lösung mit maximalen Kosten.

    (a) Definieren Sie eine Nachbarschaftsbeziehung N zwischen zwei Lösungen $O_1, O_2 \subseteq O$. Dabei soll es möglich seien zwischen zwei Lösungen O_1, O_2 zwei Objekte $o_1 \in O_1$ und $o_2 \in O_2$ auszutauschen, so dass O_1 und O_2 weiterhin Lösungen bleiben. Insbesondere soll die Nachbarschaftsbeziehung es ermöglichen die Anzahl der Objekte einer Lösung zu verändern.

(b) Entwerfen Sie eine Greedy Strategie für das Rucksackproblem. Liefert Ihr Greedy-Algorithmus stets eine Lösung mit maximalen Kosten?

(c) Entwerfen Sie eine rekursive Lösung für das Rucksackproblems. Starten Sie mit einer leeren Objektmenge und durchlaufen Sie systematisch alle möglichen Objektmengen.

68. Der *Grad* eines Knotens $v \in V$ in einem Graphen G sei die Anzahl der mit v inzidenten Kanten in G. Zeigen Sie:

(a) In einem Graphen ist die Anzahl der Knoten mit ungeradem Grad eine gerade Zahl.

(b) Jeder Graph $G = (V, E)$ mit $|V| \geq 2$ hat mindestens zwei Knoten, die denselben Grad haben.

(c) Die Anzahl der Knoten in einem Graphen, bei dem jeder Knoten den Grad 3 hat, ist eine gerade Zahl.

(d) Für jedes gerade $n \geq 4$ gibt es einen zusammenhängenden Graphen mit n Knoten, die alle den Grad 3 haben.

69. Ein ungerichteter Graph $G = (V, E)$ heisst *selbstkomplementär*,

wenn G und $\overline{G} := (V, \{\{u, v\} | u, v \in V, u \neq v, \{u, v\} \notin E\})$ sich nur in der Bezeichnung ihrer Knoten unterscheiden. Genauer bedeutet dies, dass es eine Bijektion $f : V \to V$ mit $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in \overline{E}$ gibt. G und $\overline{G}$ heissen dann auch *isomorph*, $G \cong \overline{G}$.

(a) Zeichnen Sie alle selbstkomplementären Graphen mit höchstens 5 Knoten.

(b) Zeigen Sie: Wenn ein Graph G nicht zusammenhängend ist, dann ist $\overline{G}$ zusammenhängend.

70. Zeigen Sie: Für jeden Graphen $T = (V_T, E_T)$ sind die folgenden Aussagen äquivalent.

(a) T ist ein Baum.

(b) T ist kreisfrei und $|E_T| = |V_T| - 1$.

(c) T ist zusammenhängend und $|E_T| = |V_T| - 1$.

(d) T ist zusammenhängend und jede Kante ist eine Brücke.

(e) Zwischen je zwei Knoten aus T existiert genau ein Weg.

(f) T ist kreisfrei und durch Einfügen einer Kante zwischen zwei nicht adjazenten Knoten entsteht ein Kreis.

71. Zeigen Sie: Jeder Baum $T = (V_T, E_T)$ mit $|V_T| \geq 2$ enthält mindestens zwei Knoten vom Grad 1.

72. Es sei G ein zweifachzusammenhängender planarer Graph dessen kürzester Kreis die Länge k hat.

    Zeigen Sie, dass G höchstens $(|V_G| - 2)\frac{k}{k-2}$ Kanten besitzt.

73. Beweisen Sie, dass die Graphen $K_{3,3}$ und K_5 nicht planar sind.

74. Beweisen Sie, dass der Petersen-Graph (siehe Abbildung 21.11) nicht planar ist.

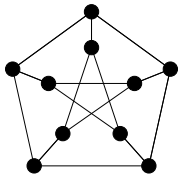


Abbildung 21.11: Petersen-Graph

75. Ist der Graph aus Abbildung 21.12 planar? Begründen Sie Ihre Antwort.

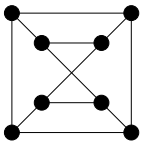


Abbildung 21.12: Graph zu Aufgabe 7

76. Sei $G = (V, E)$ ein Graph mit $|V| > 3$. Sei N_i die Anzahl der Knoten vom Grad i in G. Ein Graph heisst *maximal planar*, falls er planar ist, aber das Einfügen einer beliebigen, noch nicht vorhandenen Kante die Planarität zerstört. Zeigen Sie:

    (a) Wenn G maximal planar ist, so gilt:

    $$12 = 3N_3 + 2N_4 + N_5 - N_7 - 2N_8 - 3N_9 - 4N_{10} - \ldots$$

    (b) Wenn G maximal planar ist, so gibt es mindestens 4 Knoten vom Grad kleiner als 6. Falls G maximal planar ist und keine Knoten vom Grad 3 oder 4 hat, so hat G mindestens 12 Knoten vom Grad 5.

    (c) Falls G planar und mindestens 5-fach zusammenhängend ist, so ist $|V| \geq 12$.

    (d) Falls G planar ist, so ist G nicht 6-fach zusammenhängend.

77. Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ homeomorph. Zeigen Sie:

$$|E_1| - |V_1| = |E_2| - |V_2| \,.$$

78. Sei $G = (V, E)$ ein Graph. Der Graph $G' = (V', E')$ entsteht aus G durch *Kontraktion* der Kante $\{u, v\} \in E$, indem man u und v durch einen neuen Knoten w ersetzt, der mit allen Nachbarn von u, v in G verbunden ist, d.h.,

$$V' = V - \{u, v\} \cup \{w\}$$

und

$$E' = E - \{\{u, v\}\} \cup \{\{w, z\} \mid z \in V - \{u, v\} : \ \{v, z\} \in E \text{ oder } \{u, z\} \in E\}.$$

Ein Graph G heisst *kontrahierbar* zu einem Graphen G', wenn es eine Folge von Kantenkontraktionen gibt, die G in G' überführt.

Man zeige:

(a) Wenn G zum $K_{3,3}$ kontrahierbar ist, so enthält G einen Teilgraphen, der homeomorph zum $K_{3,3}$ ist.

(b) Wenn G zum K_5 kontrahierbar ist, so enthält G einen Teilgraphen, der homeomorph zum K_5 oder homeomorph zum $K_{3,3}$ ist.

(c) Ein Graph ist genau dann nicht-planar, wenn er einen Teilgraphen enthält, der zum $K_{3,3}$ oder zum K_5 kontrahierbar ist.

79. Man gebe einen nicht-planaren Graphen an, der weder zum $K_{3,3}$ noch zum K_5 kontrahierbar ist. Warum widerspricht dies nicht der Aufgabe 1 (c)?

80. (a) Man gebe für die Graphen, die man erhält, wenn man aus dem K_5 bzw. dem $K_{3,3}$ eine Kante entfernt, den geometrisch dualen Graphen an.

(b) Man gebe einen Graphen G mit mindestens 5 Knoten an, der ein Dual besitzt, das isomorph zu G ist.

(c) Man gebe einen Graphen an der zwei Duale besitzt, die nicht isomorph sind.

81. Welche der folgenden Graphen sind planar? Begründen Sie Ihre Antworten.

(a) K_4

(b) $K_5 - e$ (K_5 in dem eine Kante entfernt wird)

(c) $K_6 - e$

(d) $K_{3,3} - e$

(e) $K_{4,4} - e$

(f) $K_{2,10}$

(g) kreisfreie Graphen

K_n bezeichne den vollständigen Graphen $(\{v_1, \ldots, v_n\}, \{\{v_i, v_j\} \mid 1 \leq i < j \leq n\})$.

$K_{n,m}$ bezeichne den vollständig bipartiten Graphen $(\{v_1, \ldots, v_n, w_1, \ldots w_m\}, \{\{v_i, w_j\} \mid \ 1 \leq i \leq n, \ 1 \leq j \leq m\})$

82. Untersuchen Sie für welche der Graphen aus Aufgabe 13 die Eulerformel erfüllt ist.

83. Bestimmen Sie die Färbungszahl der folgenden Graphen:

    (a) C_n (Kreis mit n Knoten)

    (b) K_n

    (c) $K_{n,m}$

    (d) bipartite Graphen

    (e) kreisfreie Graphen

84. Die *Kreuzungszahl* eines Graphen G ist die minimale Anzahl paarweiser Überschneidungen von Linien, die beim Zeichnen von G in der Ebene auftreten.

    Bestimmen Sie die Kreuzungszahl der folgenden Graphen:

    (a) K_4

    (b) K_5

    (c) $K_{3,3}$

    (d) Petersen Graph (Aufgabe 6)

    (e) K_6

85. Welche der folgenden Graphklassen sind abgeschlossen bezüglich der Minorenoperation? (Eine Graphklasse $\mathcal{G}$ ist abgeschlossen bezüglich der Minorenoperation, falls mit einem Graphen G auch jeder Minor von G in $\mathcal{G}$ liegt.)

    (i) die Menge aller Graphen ohne Kreise der Länge ≤ 5,

    (ii) die Menge aller Graphen ohne Kreise der Länge ≥ 5,

    (iii) die Menge aller Graphen ohne Kreise,

    (iv) die Menge aller Bäume,

    (v) die Menge aller Graphen mit maximalem Knotengrad ≤ 4,

(vi) die Menge aller 3-färbbaren Graphen,

(vii) die Menge aller planaren Graphen,

(viii) die Menge aller Graphen, die keinen Teilgraphen enthalten, der homeomorph zum K_5 ist.

86. Für eine Menge $\mathcal{H}$ von Graphen sei

$$\text{Forb}(\mathcal{H}) := \{G \mid G \text{ ist Graph und } \forall H \in \mathcal{H} : H \text{ ist kein Minor von } G\}.$$

Zeigen Sie, dass die folgenden Graphklassen $\mathcal{G}_1$ und $\mathcal{G}_2$ abgeschlossen bezüglich der Minorenoperation sind und geben Sie möglichst kleine Mengen $\mathcal{H}_1, \mathcal{H}_2$ von verbotenen Teilgraphen mit $\mathcal{G}_1 = \text{Forb}(\mathcal{H}_1)$ und $\mathcal{G}_2 = \text{Forb}(\mathcal{H}_2)$ an.

(a) $\mathcal{G}_1 = \{G|$die Zusammengangskomponenten von G sind Kreise oder Wege$\}$

(b) $\mathcal{G}_2 = \{G \mid G \text{ ist Wald}\}$

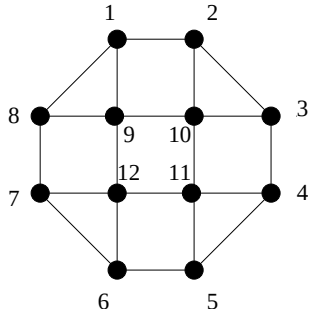
87. Seien $\mathcal{H}_1$ und $\mathcal{H}_2$ endliche Mengen von Graphen und $\mathcal{G}_1 = \text{Forb}(\mathcal{H}_1)$, $\mathcal{G}_2 = \text{Forb}(\mathcal{H}_2)$.

Zeigen Sie:

(a) $\mathcal{G}_1 \cap \mathcal{G}_2$ ist abgeschlossen bezüglich der Minorenoperation und $\mathcal{G}_1 \cap \mathcal{G}_2 = \text{Forb}(\mathcal{H}_1 \cup \mathcal{H}_2)$.

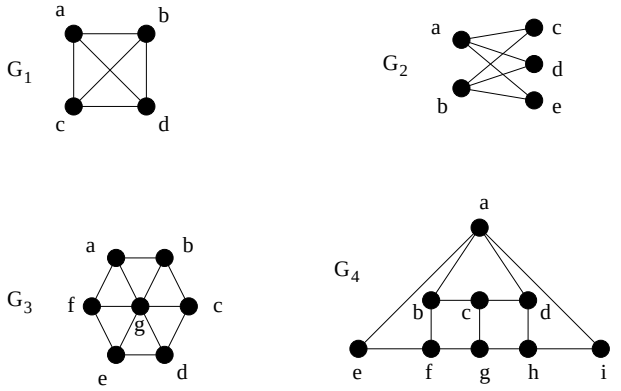
(b) $\mathcal{G}_1 \cup \mathcal{G}_2$ ist abgeschlossen bezüglich der Minorenoperation, aber es gibt nicht notwendigerweise eine Teilmenge $\mathcal{H} \subseteq \mathcal{H}_1 \cup \mathcal{H}_2$, so dass $\mathcal{G}_1 \cup \mathcal{G}_2 = \text{Forb}(\mathcal{H})$.

88. G sei der folgende Graph:



(a) Welche der unten abgebildeten Graphen G_1, G_2, G_3, G_4 sind Minor von G?

(b) Welche der unten abgebildeten Graphen G_1, G_2, G_3, G_4 enthält G als homeomorphen Teilgraphen?



89. Zeigen Sie, dass die Teilgraph-Relation auf der Menge aller Bäume keine Wohlquasiordnung ist.

90. Gegeben sei eine beliebige endliche Menge $\mathcal{X}$ von Graphen die abgeschlossen bezüglich Minoren ist. Bestimmen Sie eine endliche Menge $\mathcal{H}_\mathcal{X}$ mit $\mathcal{X} = \text{Forb}(\mathcal{H}_\mathcal{X})$.

91. Beweisen Sie, dass die Graphen $K_{2,3}$ und K_4 nicht außenplanar sind.

92. Sei G ein 2-fach zusammenhängender 2-reduzierbarer Graph mit mindestens 4 Knoten. Zeigen Sie:

    (a) G hat keinen Knoten vom Grad ≤ 1.

    (b) Nach einem 2-Reduktionsschritt ist G weiterhin 2-fach zusammenhängend und 2-reduzierbar.

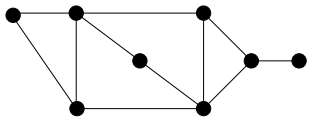
93. Sei $G = (V, E)$ ein Graph. Eine Teilmenge $V' \subseteq V$ heisst *unabhängig*, wenn für alle $u, v \in V'$ gilt, dass $\{u, v\} \notin E$. Die *Unabhängigkeitszahl* $\alpha(G)$ von G ist definiert als die Grösse einer größtmöglichen unabhängigen Menge in G.

    Geben Sie einen polynomiellen Algorithmus an, der $\alpha(G)$ für einen 2-fach zusammenhängenden, 2-reduzierbaren Graphen G berechnet.

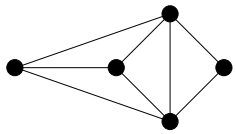
    (Hinweis: Bei der Reduktion eines Knotens vom Grad 2 speichere man an der neu entstehenden (bzw. bereits vorhandenen) Kante Informationen über eine maximale unabhängige Menge des Teilgraphen, der zu dieser Kante reduziert wurde. Die Zugehörigkeit der Endknoten der Kante zur unabhängigen Menge ist dabei zu beachten.)

94. Wenden Sie den Algorithmus zum Test auf 2-Reduzierbarkeit aus der Vorlesung auf die folgenden Graphen an.

(a)

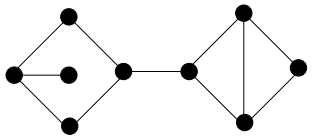


(b)

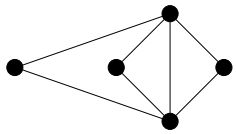


95. Wenden Sie den Algorithmus zum Test auf Außenplanrität aus der Vorlesung auf die folgenden Graphen an.

(a)



(b)



96. Geben Sie für die beiden Graphen aus Aufgabe 94 jeweils eine Baumdekomposition mit minimaler Weite an.

97. Bestimmen Sie die Baumweite der folgenden Graphen:

(a) K_n

(b) $K_{m,n}$

(c) C_n

98. Mit Baumweite(G) bezeichnen wir die Baumweite eines Graphen G.

(a) Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei homeomorphe Graphen. Zeigen Sie, dass Baumweite(G_1) = Baumweite(G_2) gilt.

(b) Sei G ein Graph und sei H ein Graph, der aus G durch eine Folge von Kantenkontraktionen hervorgeht. Zeigen Sie, dass Baumweite(G) $\geq$ Baumweite(H).

99. Sei G ein Graph mit Baumweite(G) $\leq k$. Zeigen Sie, dass es in G einen Knoten vom Grad $\leq k$ gibt.

100. Sei $k \geq 1$ und $G = (V, E)$ ein k-fach zusammenhängender Graph mit $|V| \geq k + 1$.

Zeigen Sie: Baumweite(G) $\geq k$.

101. Sei $G = (V, E)$ ein Graph mit Baumweite k und $(\mathcal{X}, T)$ eine Baumdekomposition für G der Weite k. Weiter seien $u, v \in V_T$ zwei benachbarte Knoten im Baum T. Seien $T_u = (V_{T_u}, E_{T_u})$ und $T_v = (V_{T_v}, E_{T_v})$ mit $u \in V_{T_u}$ und $v \in V_{T_v}$ die beiden Teilbäume von T, die entstehen, wenn die Kante $\{u, v\}$ entfernt wird. Sei

$$V_u := \left(\bigcup_{z \in V_{T_u}} X_z \right) - X_v \text{ und } V_v := \left(\bigcup_{z \in V_{T_v}} X_z \right) - X_u.$$

Wieviele Knoten muss der Graph G haben, damit immer

$$|V_u| < \frac{2}{3}(|V| - |X_v|) \text{ oder } |V_v| < \frac{2}{3}(|V| - |X_u|)$$

gilt?

102. Sei $G = (V, E)$ ein Graph und $(\mathcal{X}, T)$ eine Baumdekomposition für G mit $T = (V_T, E_T)$ und $\mathcal{X} = \{X_u \subseteq V \mid u \in V_T\}$.

Zeigen Sie: Wenn $G' = (V', E')$ ein zusammenhängender Teilgraph von G ist, so bilden die Knoten $\{u \in V_T \mid X_u \cap V' \neq \emptyset\}$ einen Teilbaum von T.

103. Sei $G = (V, E)$ ein zusammenhängender Graph und $(\mathcal{X}, T)$ eine Baumdekomposition für G mit $T = (V_T, E_T)$ und $\mathcal{X} = \{X_u \subseteq V \mid u \in V_T\}$. Seien $u, v \in V$ und $i, j \in V_T$, so dass $u \in X_i$ und $v \in X_j$.

Zeigen Sie: Für jeden Knoten $l \in V_T$, der auf dem Weg von i nach j in T liegt, enthält X_l mindestens einen Knoten von jedem Weg von u nach v in G.

104. Sei $G = (V, E)$ ein Graph und $v_1, v_2, v_3 \in V$, so dass v_1, v_2, v_3 eine Clique in G bilden. Sei $(\mathcal{X}, T)$ eine Baumdekomposition für G mit $T = (V_T, E_T)$ und $\mathcal{X} = \{X_u \subseteq V \mid u \in V_T\}$.

Zeigen Sie: Es gibt ein $u \in V_T$, so dass $v_1, v_2, v_3 \in X_u$.

105. Zeigen Sie: Für jede natürliche Zahl k ist die Menge der Graphen mit Wegweite höchstens k abgeschlossen bezüglich Minoren.

Wie sehen die verbotenen Minoren für die Menge der Graphen mit Wegweite 1 aus?

106. Sei

$$P : \mathcal{G} \mapsto \{\text{ja,nein}\}$$

die Eigenschaft, die beschreibt, ob ein Graph G einen Kreis der Länge genau 3 besitzt. Geben Sie einen Repräsentanten für jede Äquivalenzklasse der Äquivalenzrelation $\sim_{\Pi,2}$ bezüglich der Verknüpfung von 2-terminale Graphen durch Knotenidentifikation an.

107. Sei

$$P : \mathcal{G} \mapsto \{\text{ja,nein}\}$$

die Eigenschaft, die beschreibt, ob ein Graph G 3-färbbar ist. Beweisen Sie, dass die Äquivialenzrelation $\sim_{l,P}$ für $l = 3$, also für 3-terminale Graphen, nur endlich viele Äquivalenzklassen besitzt.

Geben Sie für jede Äquivalenzklasse einen Repräsentanten an.

108. Sei $G = (V, E)$ ein Graph. Zeigen oder widerlegen Sie, dass für jedes $l \geq 1$ die Relation $\sim_{l,P}$ für die folgenden Grapheigenschaften P endlich viele Äquivalenzklassen hat.

(a) $|V| = |E|$

(b) G hat einen Eulerweg.
Ein Eulerweg ist ein geschlossener Kantenzug, der alle Kanten aus E genau einmal durchläuft (Knoten dürfen mehrmals durchlaufen werden).

(c) $|V|$ ist durch 5 teilbar.

109. Seien $f(n)$ und $g(n)$ zwei der folgenden Funktionen.

(a) n^2 (b) n^3 (c) $n^2 \log n$

(d) 2^n (e) $n^{\log n}$ (f) n^2 falls n ungerade, 2^n sonst

Bestimmen Sie, ob (1) $f(n) \in \mathcal{O}(g(n))$, (2) $f(n) \in \Omega(g(n))$ oder (3) $f(n) \in \Theta(g(n))$ ist.

110. Betrachten Sie die Turingmaschine $M = (Q, \Sigma, \delta, s)$ mit Zustandsmenge $Q = \{s, q\}$, Bandalphabet $\Sigma = \{0, 1, \#, \triangleright\}$ und Übergangsfunktion δ wie in Abbildung 21.13. Das höchstwertigste Bit der Eingabe steht dabei immer links und es gibt keine führenden Nullen.

(a) Welche Funktion berechnet die Turingmaschine? Zeigen Sie die Funktionsweise an den Eingaben 10111 und 11111.

(b) Geben Sie eine Turingmaschine an, die bei Eingabe einer ganzen Zahl $n \geq 0$ in Binärdarstellung die Binärdarstellung von $n + 1$ berechnet.

(c) Geben Sie eine Turingmaschine an, die bei Eingabe einer ganzen Zahl $n > 0$ in Binärdarstellung die Binärdarstellung von $n - 1$ berechnet.

$p \in Q$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	0	$(s, 0, \rightarrow)$
s	1	$(s, 1, \rightarrow)$
s	#	$(q, \#, \leftarrow)$
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
q	0	$(h, 1, \perp)$
q	1	$(q, 0, \leftarrow)$
q	$\triangleright$	$(h, \triangleright, \rightarrow)$

Abbildung 21.13: Turingmaschine für Aufgabe 110

111. Definieren Sie für eine 1-Band-Turingmaschine mit beidseitig unendlichem Turingband die Begriffe Konfiguration und Berechnungsschritt. Zeigen Sie, dass alles, was mit einer beidseitig unbeschränkten Turingmaschine M mit Zeitschranke $f(n)$ berechnet werden kann, auch mit einer 2-Band-Turingmaschine M' mit Zeitschranke $g(n) \in O(f(n))$ berechnet werden kann.

112. (a) Entwerfen Sie eine 1-Band-Turingmaschine, die die Sprache $L = \{0^k 10^k \mid k \in \mathbb{N}\}$ entscheidet. Welche Zeitschranke hat Ihre Maschine?

(b) Entwerfen Sie eine 2-Band-Turingmaschine, die L mit der Zeitschranke $O(n)$ entscheidet.

113. Seien $k \geq 1$ und $l \geq 0$ natürliche Zahlen. Eine (k, l)-Turingmaschine sei eine gewöhnliche k-Band-Turingmaschine mit den folgenden Einschränkungen:

(a) Das 1. Band ist ein Eingabeband, dessen Inhalt nie verändert wird.

(b) Auf den Bändern $2, 3, \ldots, k$ werden höchstens die ersten l Felder beschrieben.

Zeigen Sie, dass jede Sprache, die von einer (k, l)-Turingmaschine entschieden werden kann, auch mit einer $(1, 0)$-Turingmaschine entschieden werden kann.

114. Zeigen Sie, dass jede nichtdeterministische Turingmaschine M durch eine nichtdeterministische Turingmaschine M' mit konstantem Zeitverlust simuliert werden kann, so dass für alle x, $M(x) = M'(x)$, aber M' in jedem Schritt nur zwei zulässige Nachfolgekonfigurationen hat.

115. (a) Zeigen Sie, dass das Problem, für eine Eingabe n, gegeben in unärer Darstellung, zu entscheiden, ob n eine Primzahl ist, in P ist.

    (b) Erläutern Sie, wie man mit einer deterministischen Turingmaschine in polynomieller Zeit zwei gegebene Binärzahlen multiplizieren kann.

    (c) Zeigen Sie, dass das Problem, für eine Eingabe n, gegeben in binärer Darstellung, zu entscheiden, ob n keine Primzahl ist, in NP ist.

    (d) In welcher Komplexitätsklasse liegt das Problem, für eine Eingabe n, gegeben in binärer Darstellung, zu entscheiden, ob n eine Primzahl ist?

116. Die Funktionen $f(n)$ und $g(n)$ seien echte Komplexitätsfunktionen. Zeigen Sie, dass dann auch

    (a) $f(n) + g(n)$

    (b) $f(n) \cdot g(n)$

    (c) $2^{g(n)}$

    (d) $f(g(n))$, falls $f(n) \geq n$

    echte Komplexitätsfunktionen sind.

117. Zeigen Sie, dass die folgenden Funktionen echte Komplexitätsfunktionen sind:

(1) $\lceil \log_2 \rceil^2(n)$	(2) $n \cdot \lceil \log_2(n) \rceil$	(3) n^2
(4) $n^3 + 3^n$	(5) 2^n	(6) 2^{n^2}
(7) $\lceil \sqrt{n} \rceil$	(8) $n!$	

118. Sei M eine $f(n)$-platzbeschränkte Turingmaschine, wobei $f(n) \geq \log n$. Zeigen Sie, dass es eine $f(n)$-platzbeschränkte Turingmaschine M' gibt, die dieselben Worte wie M akzeptiert und auf jeder Eingabe hält.

119. Seien L_1 und L_2 zwei Sprachen über Σ. Zeigen Sie: Falls $L_1 \in \mathrm{P}$ und $L_2 \neq \emptyset$ und $L_2 \neq \Sigma^*$, so ist $L_1 \leq_p L_2$.

120. Zeigen Sie, dass das folgende 3-SAT-Problem (1-IN-3-SAT) NP-vollständig ist:

    Problem: 1-IN-3-SAT:

Gegeben: Eine Menge von Variablen $X = \{x_1, \ldots, x_n\}$ und eine Menge von Klauseln $F_X = \{C_1, \ldots, C_m\}$ mit genau drei Literalen über X.

Frage: Gibt es eine Belegung $t : X \to \{T, F\}$ für X, so dass jede Klausel bzgl. t genau ein wahres Literal enthält?

121. Ist das folgende Cliquenproblem (k-Clique) in P oder NP-vollständig? Begründen Sie Ihre Antwort.

Problem: k-Clique

Gegeben: Ein Graph $G = (V, E)$.

Frage: Gibt es in G einen Teilgraphen mit k Knoten der eine Clique ist?

122. Zeigen Sie die NP-Vollständigkeit der folgenden Probleme:

(a) SHORTEST SIMPLE PATH
Gegeben: Ein ungerichteter Graph $G = (V, E)$, für jede Kante $e \in E$ ein Gewicht $c(e) \in \mathbb{Z}$, eine Zahl $k \in \mathbb{Z}$ und zwei Knoten $s, t \in V$.
Frage: Gibt es einen einfachen Weg von s nach t in G, so dass die Summe der Kantengewichte auf diesem Weg höchstens k ist. Ein einfacher Weg ist ein Weg, der jeden Knoten höchstens einmal enthält.

(b) SUBGRAPH ISOMORPHISM
Gegeben: Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$.
Frage: Enthält G_1 einen zu G_2 isomorphen Teilgraphen, d.h. existieren $V \subseteq V_1$, $E \subseteq E_1$ und eine bijektive Funktion $f : V_2 \to V$, so dass $\{u, v\} \in E_2 \Leftrightarrow \{f(u), f(v)\} \in E$ für alle $u, v \in V_2$ gilt?

(c) LARGEST COMMON SUBGRAPH
Gegeben: Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ und ein $k \in \mathbb{N}$.
Frage: Existieren zwei Teilmengen $E_1' \subseteq E_1$ und $E_2' \subseteq E_2$ mit $|E_1'| = |E_2'| \geq k$, so dass die zwei Graphen $G_1' = (V_1, E_1')$ und $G_2' = (V_2, E_2')$ isomorph sind?

(d) SETCOVER
Gegeben: Eine endliche Menge X, $\mathcal{F} \subseteq \mathcal{P}(X)$ und eine Zahl k.
Frage: Gibt es $C \subseteq \mathcal{F}$, mit $X = \cup_{S \in C} S$ und $|C| \leq k$?

123. Für zwei Wörter $u, v \in \{0,1\}^n$ der Länge n sei $u \vee v$ die bitweise Oderverknüpfung ('$0 \vee 0 = 0$', '$0 \vee 1 = 1$', '$1 \vee 0 = 1$', '$1 \vee 1 = 1$') von u und v und $u \wedge v$ sei die bitweise Undverknüpfung ('$0 \wedge 0 = 0$', '$0 \wedge 1 = 0$', '$1 \wedge 0 = 0$', '$1 \wedge 1 = 1$') von u und v.

Zeigen Sie, dass die folgende Sprache $L \subseteq \{0, 1, \triangleright', \triangleright'', \triangleright'''\}$ in NP ist.

L sei die Menge aller Wörter

$x = \triangleright' u_1 \triangleright' u_2 \triangleright' \ldots \triangleright' u_k \triangleright'' v_1 \triangleright' v_2 \triangleright' \ldots \triangleright' v_l \triangleright''' w_1 \triangleright' w_2 \triangleright' \ldots \triangleright' w_m$

mit $u_i, v_i, w_i \in \{0,1\}^n$, für ein $n \geq 0$ und der folgenden Eigenschaft.

Es existieren $r \leq k$ Indices $i_1, \ldots, i_r \in \{1, \ldots, k\}$, so dass

jedes Wort $u_{i_1} \vee \cdots \vee u_{i_r} \wedge v_j$ für $j = 1, \ldots, l$ höchstens eine 1 und

jedes Wort $u_{i_1} \vee \cdots \vee u_{i_r} \wedge w_i$ für $i = 1, \ldots, m$ mindestens eine 1 enthält.

124. Zeigen Sie, dass die Sprache L aus Aufgabe 8 NP-schwer ist.

125. Seien L_1 und L_2 zwei Sprachen über Σ. Zeigen Sie: Falls $L_1 \in \mathrm{P}$ und $L_2 \neq \emptyset$ und $L_2 \neq \Sigma^*$, so ist $L_1 \leq_p L_2$.

126. Betrachten Sie das Problem 2-SAT (Analog zu 3-SAT definiert, aber jede Klausel enthält genau zwei Literale.) und zeigen Sie, dass 2-SAT in polynomieller Zeit gelöst werden kann.

127. Zeigen Sie, dass das folgende 3-SAT-Problem (1-IN-3-SAT) NP-vollständig ist:

    Problem: 1-IN-3-SAT:

    Gegeben: Eine Menge von Variablen $X = \{x_1, \ldots, x_n\}$ und eine Menge von Klauseln $F_X = \{C_1, \ldots, C_m\}$ mit genau drei Literalen über X.

    Frage: Gibt es eine Belegung $t : X \to \{T, F\}$ für X, so dass jede Klausel bzgl. t genau ein wahres Literal enthält?

128. Gegeben sei ein gerichteter Graph $G = (V, E)$ und zwei Abbildungen $f : V \to \mathbb{N}_0$ und $g : E \to \mathbb{N}_0$.

    (a) Zeigen Sie, dass das Problem, zu entscheiden, ob es einen Weg zwischen zwei Knoten in G gibt, der jede Kante $e \in E$ genau $g(e)$ mal enthält, in P liegt.

    (b) Zeigen Sie, dass das Problem, zu entscheiden, ob es einen Weg zwischen zwei Knoten in G gibt, der jeden Knoten $u \in V$ genau $f(u)$ mal enthält, in NP liegt.

129. Ist das Cliquenproblem für konstante Cliquengröße k in P oder NP-vollständig? Begründen Sie Ihre Antwort.

130. Zeigen Sie die NP-Vollständigkeit der folgenden Probleme:

(a) SHORTEST SIMPLE PATH
Gegeben: Ein ungerichteter Graph $G = (V, E)$, für jede Kante $e \in E$ ein Gewicht $c(e) \in \mathbb{Z}$, eine Zahl $k \in \mathbb{Z}$ und zwei Knoten $s, t \in V$.
Frage: Gibt es einen einfachen Weg von s nach t in G, so dass die Summe der Kantengewichte auf diesem Weg höchstens k ist. Ein einfacher Weg ist ein Weg, der jeden Knoten höchstens einmal enthält.

(b) SUBGRAPH ISOMORPHISM
Gegeben: Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$.
Frage: Enthält G_1 einen zu G_2 isomorphen Teilgraphen, d.h. existieren $V \subseteq V_1$, $E \subseteq E_1$ und eine bijektive Funktion $f : V_2 \longrightarrow V$, so dass $(u, v) \in E_2 \Leftrightarrow (f(u), f(v)) \in E$ für alle $u, v \in V_2$ gilt?

(c) LARGEST COMMON SUBGRAPH
Gegeben: Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ und ein $k \in \mathbb{N}$.
Frage: Existieren zwei Teilmengen $E_1' \subseteq E_1$ und $E_2' \subseteq E_2$ mit $|E_1'| = |E_2'| \geq k$, so dass die zwei Graphen $G_1' = (V_1, E_1')$ und $G_2' = (V_2, E_2')$ isomorph sind?

131. Das Problem 3-PARTITION ist wie folgt definiert:

Gegeben: Eine Folge $a_1, \ldots, a_{3n}$ von natürlichen Zahlen und eine natürliche Zahl B mit $0 < a_i < B$ für $i = 1, \ldots, 3n$ und $\sum_{i=1}^{3n} a_i = nB$.

Frage: Gibt es paarweise disjunkte 3-elementige Teilmengen $A_1, \ldots, A_n$, $A_i \subseteq \{1, \ldots, 3n\}$ mit $\bigcup_{i=1}^{n} A_i = \{1, \ldots, 3n\}$, so dass für $i = 1, \ldots, n$

$$\sum_{j \in A_i} a_j = B \quad ?$$

In der Vorlesung wurde angegeben, dass 3-PARTITION NP-vollständig ist. Man zeige, dass das Problem auch NP-vollständig ist, wenn man nicht verlangt, dass die Mengen A_i 3-elementig sind. Dabei kann die NP-Vollständigkeit von 3-PARTITION vorausgesetzt werden.

132. Man zeige, dass das folgende Problem NP-vollständig ist.

Gegeben: Eine $n \times m$-Matrix $A = (a_{ij})$ mit $a_{ij} \in \{0, 1\}$, zwei Zahlen $n' \leq n$ und $m' \leq m$ und ein Wert L.

Frage: Gibt es n' verschiedene Zeilen $z_1, \ldots, z_{n'}$, $1 \leq z_i \leq n$ und m' verschiedene Spalten $s_1, \ldots, s_{m'}$, $1 \leq s_i \leq m$, so dass die Matrix, die nur aus diesen Zeilen und Spalten besteht, mindestens L Einsen enthält, d.h.,

$$\sum_{i=1}^{n'} \sum_{j=1}^{m'} a_{z_i s_j} \geq L \quad ?$$

(Hinweis: Reduktion von CLIQUE.)

133. PARTITION in 3 Mengen:

Gegeben: Eine Folge $a_1, \ldots, a_n$ von natürlichen Zahlen und eine natürliche Zahl B mit $0 < a_i < B$ für $i = 1, \ldots, n$ und $\sum_{i=1}^{n} a_i = 3B$.

Frage: Gibt es paarweise disjunkte Teilmengen $A_1, A_2, A_3 \subseteq \{1, \ldots, n\}$ mit $A_1 \cup A_2 \cup A_3 = \{1, \ldots, n\}$, so dass für $i = 1, 2, 3$

$$\sum_{j \in A_i} a_j = B$$

gilt?

Das obige Problem ist NP-vollständig (soll nicht gezeigt werden). Man zeige, dass das Problem nicht streng NP-vollständig ist, indem man einen pseudo-polynomiellen Algorithmus für das Problem angibt.

134. Ist das folgende Problem in P oder NP-vollständig? Beweisen Sie Ihre Antwort.

Gegeben: Ein ungerichteter Graph $G = (V, E)$.

Frage: Gibt es eine Teilmenge der Knoten $V' \subseteq V$, so dass die Anzahl der Kanten in E, die mindestens einen Endknoten aus V' enthalten genau so groß ist, wie die Anzahl der Kanten, die mindestens einen Endknoten aus $V - V'$ enthalten?

135. Zeigen Sie dass das folgende Problem NP-vollständig ist:

SETCOVER
Gegeben: Eine endliche Menge X, $\mathcal{F} \subseteq \mathcal{P}(X)$ und eine Zahl k.

Frage: Gibt es $C \subseteq \mathcal{F}$, mit $X = \cup_{S \in C} S$ und $|C| \leq k$?

136. Zeigen Sie dass das folgende Problem NP-vollständig ist:

0,1 Integer Programming
Gegeben: Eine endliche Menge X von Paaren $(\overline{x}, b)$, wobei $\overline{x}$ ein m-Tupel über $\{0, 1\}$ und $b \in \{0, 1\}$ ist, ein m-Tupel $\overline{c}$ über $\{0, 1\}$ und eine ganze Zahl B.

Frage: Gibt es ein m-Tupel $\overline{y}$ ganzer Zahlen, so dass $\overline{x} \cdot \overline{y} \leq b$ für alle $(\overline{x}, b) \in X$ und $\overline{c} \cdot \overline{y} \geq B$? (Das Produkt $\overline{u} \cdot \overline{v}$ zweier m-Tupel $\overline{u} = (u_1, u_2, \ldots, u_m)$ und $\overline{v} = (v_1, v_2, \ldots, v_m)$ ist $\sum_{i=1}^{m} u_i \cdot v_i$.)

137. **Δ-Travelling Salesman Problem (Δ-TSP)**

Gegeben: Eine $n \times n$ Matrix d mit nichtnegativen Einträgen (Distanzen) die die Dreiecksungleichung erfüllen (d.h. für alle $i_1, i_2, i_3 \in \{1, \ldots, n\}$: $d(i_1, i_2) + d(i_2, i_3) \geq d(i_1, i_3)$).

Gesucht: Eine Folge $i_1, \ldots, i_m \in \{1, \ldots, n\}$ von Indices in der jeder Index $1, \ldots, n$ mindestens einmal vorkommt, so dass $d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{m-1}, i_m) + d(i_m, i_1)$ minimal ist.

(a) Zeigen Sie, dass das Δ-Travelling Salesman Problem (Δ-TSP) NP-vollständig ist.

(b) Man gebe für das Δ-Travelling Salesman Problem (Δ-TSP), einen polynomiellen Approximationsalgorithmus A mit $R_A \leq 2$ an.

Hinweis: Es kann dabei vorausgesetzt werden, dass minimale Spannbäume in einem Graphen in polynomieller Zeit berechnet werden können.

Ein Teilgraph $G' = (V, E')$ eines Graphen $G = (V, E)$ heißt *Spannbaum*, wenn G' zusammenhängend und kreisfrei ist. Ein *minimaler* Spannbaum eines Graphen G mit Kantengewichten $d(e) \in \mathbb{Z}$ für $e \in E$ ist ein Spannbaum, dessen Summe der Kantengewichte unter allen Spannbäumen von G minimal ist.

138. Betrachten Sie zu einem Optimierungsproblem A den Spezialfall, bei dem das Optimierungsziel durch eine Konstante beschränkt wird. Einen solchen Spezialfall nennt man konstante Beschränkung von A. (Man möchte also entscheiden, ob es eine Lösung gibt, die höchstens (bzw. mindestens) den Wert dieser Konstante besitzt.)

Geben Sie für die folgenden Probleme an, ob (bzw. für welche Konstanten oder unter welchen Bedingungen) die konstanten Beschränkungen in polynomieller Zeit lösbar oder NP-vollständig sind?

(a) TSP

(b) MINIMUM-COLORING

(c) BIN PACKING

(d) KNAPSACK

139. Zeigen Sie:

Es gibt keinen polynomiellen Approximationsalgorithmus A für das Cliquenproblem mit $|A(I) - OPT(I)| \leq k$ für eine Konstante k, falls P $\neq$ NP.

140. Das Färbungsproblem k-COLORABILITY für Graphen ist wie folgt definiert.

INSTANZ: Graph $G = (V, E)$ und eine natürliche Zahl $k \leq |V|$.

FRAGE: Ist der Graph mit k Farben färbbar? D.h., gibt es eine Funktion $f : V \longrightarrow \{1, \ldots, k\}$, so dass $f(u) \neq f(v)$ für $\{u, v\} \in E$ gilt?

In der Optimierungsversion COLORABILITY ist das minimale k zu berechnen, so dass G mit k Farben gefärbt werden kann. In der Vorlesung ist gezeigt worden, dass es keinen Approximationsalgorithmus A mit $R_A < \frac{4}{3}$ geben kann, falls P $\neq$ NP.

Zeigen Sie: Es gibt keinen polynomiellen Approximationsalgorithmus A für COLORABILITY mit $R_A^\infty < \frac{4}{3}$, falls P $\neq$ NP.

Hinweis: 3-COLORABILITY ist NP-vollständig.

141. Eine *k-Aufteilung* für n Zahlen $a_1, \ldots, a_n \in \mathbb{N}$ ist eine paarweise disjunkte vollständige Aufteilung von $I = \{1, \ldots, n\}$ in k Mengen $S_1, \ldots, S_k$, so dass

$$\sum_{j \in S_i} a_j = \frac{\sum_{r=1}^{n} a_r}{k}, i = 1, \ldots, k.$$

Betrachten Sie das folgende Problem k-AUFTEILUNG.

INSTANZ: $a_1, \ldots, a_n, k \in \mathbb{N}$

FRAGE: Gibt es eine k-Aufteilung für $a_1, \ldots, a_n$?

(a) Zeigen Sie, dass das Problem k-AUFTEILUNG stark NP-vollständig ist.

(b) Zeigen Sie, dass das Problem k-AUFTEILUNG für jedes feste $k \geq 2$ schwach NP-vollständig ist.
Hinweis: Reduktion von 1-IN-3SAT (wie beim Problem KNAPSACK)

142. Zeigen Sie:

Wenn es einen polynomiellen Approximationsalgorithmus A für das Cliquenproblem mit $R_A^\infty < \infty$ gibt, dann gibt es auch ein polynomielles Approximationsschema für das Cliquenproblem.

143. Zeigen Sie:

Wenn es einen polynomiellen Approximationsalgorithmus A für VERTEX COVER mit $R_A^\infty < \infty$ gibt, dann gibt es auch ein polynomielles Approximationsschema für VERTEX COVER.

144. Zeigen Sie:

Es gibt keinen polynomiellen Approximationsalgorithmus A für das VERTEX COVER Problem mit $|A(I) - OPT(I)| \leq k$ für eine Konstante k, falls P $\neq$ NP.

145. MAX-3-SAT ist das Optimierungsproblem, bei dem zu einer gegebenen logischen Formel, in der jede Klausel 3 Literale enthält, eine Variablenbelegung gefunden werden soll, die die maximal mögliche Anzahl von Klauseln erfüllt, auch wenn die Formel selbst nicht erfüllbar ist.

Geben Sie einen Approximationsalgorithmus A für MAX-3-SAT mit $R_A \leq 2$ an.

Aufgaben zu Algorithmische Geometrie

1. Zeigen Sie, dass man mit der in der Vorlesung angegebene Ungleichung zur Erkennung einer Linksdrehung tatsächlich Linksdrehungen erkennen kann.

2. Geben Sie einen Algorithmus an, der die konvexe Hülle für eine Menge von Punkten $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^3$ berechnet. Welche Laufzeit und welchen Speicherplatz benötigt Ihr Algorithmus?

3. Gegeben seien vier Punkte q_1, q_2, p_1, p_2 in der Ebene.

   Geben Sie einen Algorithmus an, der entscheidet, ob sich die Segmente $\overline{q_1q_2}$ und $\overline{p_1p_2}$ schneiden. Der Algorithmus soll möglichst wenige arithmetische Operationen (Addition, Subtraktion, Multiplikation, Division,...) und möglichst wenige Vergleiche enthalten.

   Geben Sie außerdem einen Algorithmus an, der den minimalen Abstand des Punktes p_1 zu den Punkten des Liniensegmentes $\overline{q_1q_2}$ berechnet, also

   $$\min\{\ d(p1, q) \mid q \in \overline{q_1q_2}\ \},$$

   wobei $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$ der euklidische Abstand ist.

4. Ein Punkt $(x, y) \in \mathbb{R}^2$ liegt in der ϵ-Nähe von einem Punkt $(x', y') \in \mathbb{R}^2$, falls $|x - x'| \leq \epsilon$ und $|y - y'| \leq \epsilon$ ist.

   Gegeben sei eine Menge von Punkten $P = \{p_1, \ldots, p_n\}$ mit $p_i \in \mathbb{R}^2$, ein $\epsilon > 0$ und Anfragen der folgenden Art:

   Gegeben: Ein Punkt $(x, y) \in R^2$.
   Frage: Gibt es einen Punkt aus P in der ϵ-Nähe von (x, y)?

Es soll in einer Vorverarbeitungsphase eine Datenstruktur aufgebaut werden, so dass Anfragen der obigen Art möglichst schnell beantwortet werden können.

Erläutern sie den Aufbau der gesuchten Datenstruktur einschließlich dem notwendigen Zeit- und Platzaufwand, und wie die Anfragen damit effizient beantwortet werden können. Mit welchem Zeitaufwand können die Anfragen beantwortet werden?

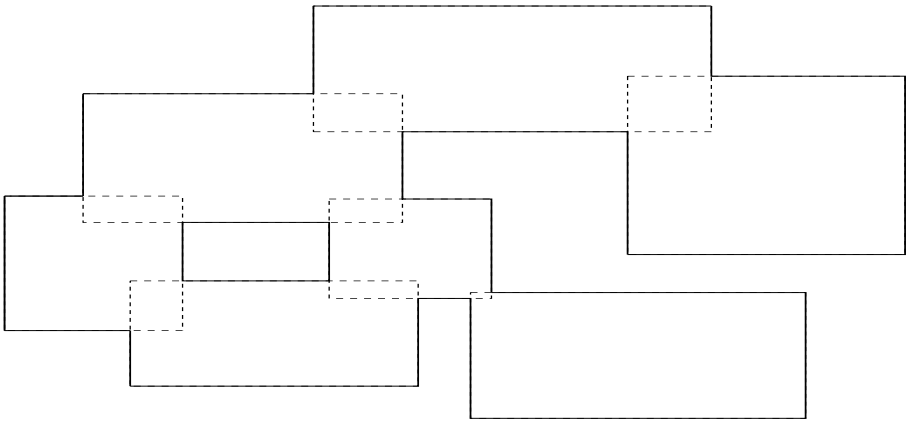
Hinweis: Verwenden Sie ein Scan-Line verfahren.

5. Zwei Liniensegmente s und s' in einer Menge horizontaler Liniensegmente sind gegenseitig sichtbar, wenn es eine vertikale Gerade gibt, die s und s', aber kein weiteres Liniensegment der Menge zwischen s und s' schneidet.

   Geben Sie einen Scan-Line Algorithmus an, der zu einer gegebenen Menge horizontaler Liniensegmente alle Paare gegenseitig sichtbarer Liniensegmente ausgibt. Sie können voraussetzen, dass alle x-Koordinaten der Anfangs- und Endpunkte paarweise verschieden sind.

6. (Ottmann/Widmayer, 4.Auflage, Seite 528) Entwerfen Sie einen möglichst effizienten Algorithmus zur Lösung des folgenden Problems:

   Gegeben sei eine Menge von n Rechtecken in der Ebene. Es ist der Umfang der von den Rechtecken gebildeten Polygone zu bestimmen. Unter Umfang sei die Länge des Randes einschließlich der entstehenden Löcher verstanden, siehe Beispiel.



Die Länge der durchgezogenen Linien ist zu berechnen.

Analysieren Sie die Komplexität des von Ihnen verwendeten Verfahrens.

Hinweis: Sie können davon ausgehen, dass die x-Koordinaten der vertikalen Seiten und die y-Koordinaten der horizontalen Seiten jeweils paarweise verschieden sind. Es ist ratsam zwei Scan-Line Durchläufe zu verwenden, womit man eine Laufzeit von $O(n \log n)$ erreichen kann.

7. In dieser Aufgabe soll ein Algorithmus entwickelt werden, der nach der Divide & Conquer Methode zu einer gegebenen Menge $P = \{p_1, \ldots, p_n\}$ von Punkten in der Ebene die konvexe Hülle $\mathrm{CH}(P)$ berechnet. Sie können voraussetzen, dass alle x- und y-Koordinaten der Punkte paarweise verschieden sind und dass die Punkte $p_1, \ldots, p_n$ aufsteigend nach x-Koordinaten sortiert sind. Benutzen Sie folgendes Programmgerüst.

   divide: (1) teile P auf in $P_1 = \{p_1, \ldots, p_{\lfloor \frac{n}{2} \rfloor}\}$ und $P = \{p_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, p_n\}$

   conquer: (2) berechne die konvexe Hülle $\mathrm{CH}(P_1)$ der Menge P_1
   (3) berechne die konvexe Hülle $\mathrm{CH}(P_2)$ der Menge P_2

   merge: (4) ...

   Beschreiben Sie, wie durch Mischen der konvexen Hüllen $\mathrm{CH}(P_1)$ und $\mathrm{CH}(P_2)$ im Schritt (4) die konvexe Hülle der Punktmenge P effizient berechnet werden kann.

   Welche Laufzeit hat ihr Algorithmus?

8. Gesucht ist das dichteste Paar von Punkten aus einer gegebenen Menge von Punkten.

   (a) Spezifizieren Sie dieses Problem formal.

   (b) Entwerfen Sie einen Algorithmus der das Problem mittels Divide and Conquer möglichst effizient löst.

   (c) Zeigen Sie die Korrektheit ihrer Lösung. (Hinweis: Zeigen Sie, dass es in einem Rechteck mit Kantenlänge ϵ und 2ϵ höchstens sechs Punkte geben kann, die einen Abstand von mindestens ϵ haben.)

   (d) Bestimmen Sie die Laufzeit ihrer Lösung.

   (e) Zeigen Sie die Optimalität ihrer Lösung. (Hinweis: benutzen Sie das Element-Uniqueness-Problem)

9. (Ottmann/Widmayer, 4.Auflage, Seite 529) Das Slot-Assignment-Problem ist wie folgt definiert:

   Gegeben sind n horizontale Liniensegmente in der Ebene.

   Jedem horizontalen Liniensegment soll eine positive ganze Zahl zugeordnet werden, so dass für jede vertikale Linie v, die Zahlen der von v geschnitten horizontalen Liniensegmente aufsteigend sortiert sind. Es sollen jedoch insgesamt möglichst wenige verschiedene Zahlen verwendet werden.

Geben Sie ein allgemeines Verfahren zur Lösung des Slot-Assignment-Problems an und analysieren Sie die Laufzeit Ihres Verfahrens.

Hinweis: Es ist möglich das Slot-Assignment Problem für eine Menge von n Segmenten in Zeit $O(n \log n)$ und Platz $O(n)$ zu lösen.

10. Gegeben sei die Menge

$$P = \{(2,8), (0,0), (6,4), (14,8), (20,6), (12,2), (18,1)\}$$

von sieben Punkten in der Ebene.

(a) Konstruieren Sie das Voronoi-Diagramm für diese Punktmenge.

(b) Bestimmen Sie ein Paar p_1, p_2 von Punkten aus P mit minimaler Distanz.

(c) Bestimmen Sie einen minimalen Spannbaum für P. Die Länge einer Kante ist die euklidische Distanz der beiden Endpunkte.

11. Gegeben seien n Punkte in der Ebene und das zugehörige Voronoi-Diagramm. Zeigen Sie, dass eine beliebige Gerade maximal $n - 1$ Kanten des Voronoi-Diagramms schneidet.

12. Wir betrachten n Geraden in der Ebene, die in allgemeiner Lage liegen sollen, d.h. keine drei Geraden schneiden sich in einem Punkt und keine Geraden sind parallel.

(a) Zeigen Sie, dass sich die Geraden in genau $\binom{n}{2}$ Punkten schneiden.

(b) Zeigen Sie, dass die Geraden die Ebene in genau $\binom{n+1}{2} + 1$ Gebiete unterteilen. Sie können voraussetzen, dass es keine vertikalen Geraden gibt.

(c) Berechnen Sie die Anzahl der Kanten (d.h. die Anzahl der Liniensegmente zwischen zwei Schnittpunkten bzw. der Halbgeraden, auf denen sich kein Schnittpunkt befindet), die sich durch die n Geraden ergeben.

13. Geben Sie einen Algorithmus an, der zu einer gegebenen Menge von Punkten in linearer Zeit die konvexe Hülle berechnet, falls das Voronoi-Diagramm der Punkte schon vorliegt.

14. Zeigen Sie mit Hilfe der Eulergleichung für planare Graphen, dass jedes Voronoi-Diagramm einer Menge von n Punkten höchstens $2 \cdot n - 4$ Knoten und höchstens $3 \cdot n - 6$ Kanten besitzt.

15. Gegeben seien zwei konvexe Polygone P_1 und P_2, die nicht überlappend sind. Die Kontur der konvexen Hülle der Eckpunkte dieser Polygone enthält genau zwei Liniensegmente, die nicht in P_1 oder P_2 enthalten sind. (Die Endpunkte dieser Liniensegmente sind die Tangentialpunkte.)

Entwerfen Sie einen Algorithmus, der diese beiden Segmente berechnet. Geben Sie die Laufzeit ihrer Lösung an.

16. In dem in der Vorlesung vorgestellten Algorithmus zur Berechnung des Voronoi-Diagramms einer Punktmenge wird die Mittelsenkrechte von zwei Punkten bestimmt. Überlegen Sie sich, wie man die Mittelsenkrechte berechnen und wie man sie angeben kann.

17. Bei der Erstellung von Voronoi-Diagrammen werden Halbgeraden verwendet. Überlegen Sie sich eine sinnvolle Repräsentation von Halbgeraden im Rechner. Geben Sie eine Berechnungsmöglichkeit für Schnittpunkte von Halbgeraden an.

18. Ein Graph heißt maximal planar, falls er planar ist, aber das Einfügen einer beliebigen, noch nicht vorhandenen Kante die Planarität zerstört. Zeigen Sie:

    (a) In jedem planaren Graphen haben mehr als die Hälfte der Knoten einen Grad kleiner als 12.

    (b) Ist ein Graph maximal planar, so gibt es mindestens 4 Knoten vom Grad kleiner als 6.

    (c) Wenn ein Graph maximal planar ist und keine Knoten vom Grad 3 oder 4 besitzt, so hat er mindestens 12 Knoten vom Grad 5.

19. Gegeben sei das folgende y-monotone Polygon (siehe Abbildung). Führen Sie für dieses Polygon den in der Vorlesung vorgestellten Algorithmus zur Triangulierung von y-monotonen Polygonen durch.

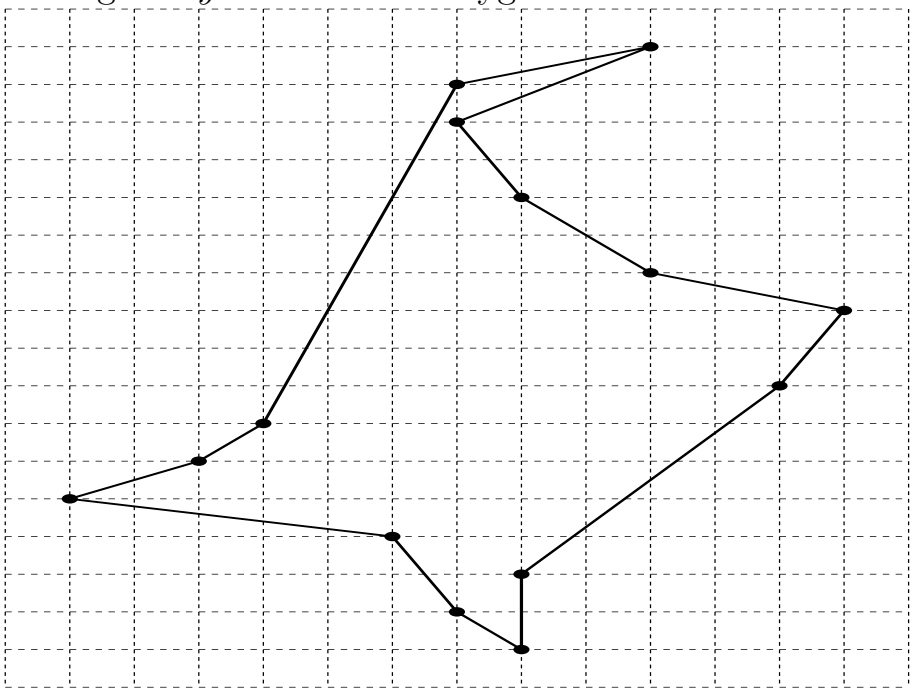


Abbildung zu Aufgabe 19

20. Häufig will man nicht ein Polygon zerlegen, sondern Punkte so durch Kanten verbinden, dass (nur) Dreiecke entstehen.

Problemstellung:
Gegeben: eine endliche Menge P von Punkten in der Ebene
Gesucht: eine Zerlegung der konvexen Hülle von P in Dreiecke, so dass die Punkte von P die Eckpunkte der Dreiecke sind.

Es gibt verschiedene Optimalitätskriterien für Dreieckszerlegungen, unter anderem:

- minimale Kantenlängensumme
- Max-Min-Winkelkriterium: der kleinste vorkommende Dreieckswinkel wird maximiert
- Min-Max-Winkelkriterium: der größte vorkommende Dreieckswinkel wird minimiert

(a) Bestimmen Sie für die folgende Punktemenge (siehe Abbildung) eine Triangulierung mit minimaler Kantenlänge.

(b) Geben Sie für jedes Paar aus obigen Optimalitätskriterien eine Punktmenge an, für die die beiden Kriterien jeweils zu unterschiedlichen Triangulierungen führen.

(c) Zeigen Sie für Punktmengen, die vier Punkte enthalten, dass die Delaunay-Triangulierung das Max-Min-Winkelkriterium erfüllt.

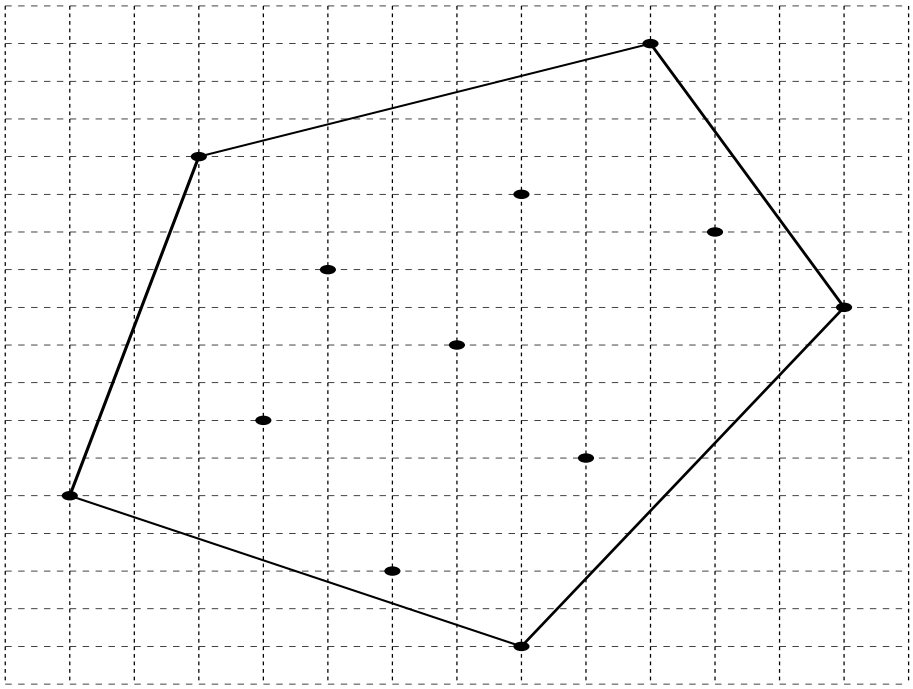


Abbildung zu Aufgabe 20

21. (a) Geben Sie einen Algorithmus an, der die 3-Färbung eines trianguliertenen einfachen Polygons berechnet. Der Algorithmus sollte eine lineare Laufzeit haben.

    (b) Kann man mit dem Algorithmus aus a) feststellen, ob beliebige Triangulierungen von Punktmengen 3-färbbar sind?

22. (Ottmann/Widmayer, 4.Auflage, Seite 528) Gegeben sei die Menge $\{A, B, C, D, E, F\}$ von Intervallen mit

$$A = [2, 3],\ B = [5, 9],\ C = [1, 4],\ D = [3, 7],\ E = [6, 8] \text{ und } F = [8, 10].$$

    (a) Geben Sie einen Intervallbaum möglichst geringer Höhe zur Speicherung dieser Intervallmenge an.

    (b) Führen Sie eine Aufspießanfrage für den Punkt $x = 3$ durch und geben Sie an, in welcher Reihenfolge die aufgespießten Intervalle entdeckt werden (ausgehend vom Intervallbaum aus a)).

23. (Ottmann/Widmayer, 4.Auflage, Seite 528) Bei der im Ottmann/Widmayer, 4.Auflage, Abschnitt 7.4.2, vorgestellten Version von Segment-Bäumen waren die Knotenlisten nicht-sortierte, doppelt verkettete Listen von Intervallnamen. Zusätzlich wurde (um das Entfernen von Intervallnamen zu unterstützen) ein separates Wörterbuch für alle Intervalle aufrechterhalten.

    Überlegen Sie sich eine andere, möglichst effiziente Möglichkeit zur Organisation der Knotenlisten, die es erlaubt auf das zusätzliche Wörterbuch zu verzichten. Geben Sie eine möglichst genaue Abschätzung der Worst-Case-Laufzeit der Einfüge- und Entferne-Operation in ihrer Datenstruktur an.

24. Gegeben sei die folgende Menge von Punkten in der Ebene:

$$(3, 7)\ (4, 2)\ (5, 8)\ (2, 1)\ (1, 4)\ (6, 3)\ (7, 9)\ (8, 5)$$

    (a) i. Fügen Sie die Punkte der Reihe nach in das anfangs leere Skelett eines Prioritäts-Suchbaumes ein.

    ii. Bestimmen Sie die Menge aller Punkte im Bereich $3 \leq x \leq 6$ und $y \leq 5$ durch eine Bereichsanfrage im Prioritäts-Suchbaum.

    iii. Entfernen Sie die Punkte in der umgekehrten Reihenfolge aus dem Prioritäts-Suchbaum.

    (b) i. Fügen Sie die Punkte der Reihe nach in einen anfangs leeren dynamischen Prioritäts-Suchbaumes ein.

    ii. Bestimmen Sie die Menge aller Punkte im Bereich $3 \leq x \leq 6$ und $y \leq 5$ durch eine Bereichsanfrage im Prioritäts-Suchbaum.

iii. Entfernen Sie die Punkte in der umgekehrten Reihenfolge aus dem Prioritäts-Suchbaum.

25. Gegeben sei ein einfaches Polygon P und eine Gerade g. Entwerfen Sie einen möglichst effizienten Algorithmus, der feststellt, ob das Polygon P monoton bezüglich der Geraden g ist. Welche Laufzeit hat ihre Lösung?

26. Gegeben sei ein einfaches Polygon P und eine doppelt-verkettete Kantenliste D, die P in monotone Subpolygone unterteilt. (D ist die Ausgabe des in der Vorlesung vorgestellten Algorithmus MakeMonotone(P).) Entwerfen Sie einen Algorithmus, der diese Subpolygone ausgibt, z.B. als Liste von Punkten. Die Laufzeit der Lösung sollte linear sein.