Effiziente Algorithmen

Master of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WiSe 2024/25

Übersicht

Inhalt

- Einleitung / Wiederholung ALD und THI
- Exakte Algorithmen für schwere Probleme
- Entwurfsmethoden
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Algorithmen für moderne Hardware
- Amortisierte Laufzeitanalysen
- Algorithmen für geometrische Probleme

Algorithmen und Datenstrukturen

- T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen.* Springer Spektrum
- T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms.* MIT Press
- Robert Sedgewick: *Algorithms.* Addison-Wesley
- Uwe Schöning: *Algorithmik.* Spektrum Akademischer Verlag
- Uwe Schöning: *Algorithmen - kurz gefasst.* Spektrum Akademischer Verlag
- Volker Heun: *Grundlegende Algorithmen.* Vieweg Verlag
- A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Datastructures and Algorithms.* Addison-Wesley
- Jon Kleinberg, Éva Tardos: *Algorithm Design.* Pearson-Addison Wesley
  `https://www.cs.princeton.edu/~wayne/kleinberg-tardos`

spezielle Themen

- F. Gurski, I. Rothe, J. Rothe, E. Wanke: *Exakte Algorithmen für schwere Graphenprobleme.* Springer Verlag
- Shimon Even: *Graph Algorithms.* Computer Science Press
- Uwe Schöning: *Theoretische Informatik - kurz gefasst.* Spektrum Akademischer Verlag
- C.H. Papadimitriou: *Computational Complexity.* Addison-Wesley
- Juraj Hromkovič: *Randomisierte Algorithmen.* Vieweg + Teubner Verlag
- Rolf Wanka: *Approximationsalgorithmen.* Teubner B.G.
- I. Gerdes, F. Klawonn, R. Kruse: *Evolutionäre Algorithmen.* Vieweg Verlag

Übersicht

Übersicht

Welche Eigenschaften interessieren uns?

- *Korrektheit:* Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen. (E. Dijkstra)
- *Laufzeit:* Wie viele Operationen werden zur Ausführung des Algorithmus auf einer idealisierten Maschine benötigt?
- *Speicherplatz:* Wie viel Speicherplatz wird benötigt?
- *Kommunikationszeit:* Bei parallelen oder verteilten Algorithmen sollen die Prozesse bzw. Threads das eigentliche Problem lösen und möglichst wenig Zeit mit Kommunikation vergeuden.
- *Güte:* Manche Probleme sind so schwer, dass sie sich nicht exakt lösen lassen bzw. eine exakte Lösung zu viel Zeit benötigen würde. Für solche Probleme ist man an möglichst guten Lösungen interessiert.

Definition: Für eine gegebene Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ bezeichnet $\mathcal{O}(g)$ die Menge der Funktionen, die asymptotisch höchstens so stark wachsen wie g.

$\longrightarrow g$ ist obere Schranke!

$$\mathcal{O}(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

Wir betrachten nur Funktionen $f : \mathbb{N}_0 \to \mathbb{N}$, da sowohl die Größe der Eingabe als auch die Anzahl der ausgeführten Operationen eines Algorithmus immer ganzzahlig und nicht-negativ sind.

Definition: Für eine gegebene Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ bezeichnet $\Omega(g)$ die Menge der Funktionen, die asymptotisch mindestens so stark wachsen wie g.

$\longrightarrow g$ ist untere Schranke!

$$\Omega(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : c \cdot g(n) \leq f(n)\}$$

Wir suchen natürlich immer die kleinste, obere Schranke für $\mathcal{O}$ sowie die größte, untere Schranke für Ω.

Definition: Für eine gegebene Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ bezeichnet $\Theta(g)$ die Menge der Funktionen, die asymptotisch genauso stark wie g wachsen.

$$\Theta(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \in \mathcal{O}(g) \ \wedge \ f \in \Omega(g)\}$$

In der Mathematik findet man Aussagen der Art

$$\frac{1}{n^2} \in \mathcal{O}\left(\frac{1}{n}\right).$$

Bei der Laufzeitanalyse von Algorithmen kommen monoton fallende Funktionen nicht vor, da Algorithmen bei größerer Eingabe nicht kürzer laufen. Obwohl es Einbrüche bei der Laufzeit geben kann.

Wir schreiben abkürzend:

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

Da wir nur Funktionen $f, g : \mathbb{N}_0 \to \mathbb{N}_0$ betrachten, sind einige der obigen Schreibweisen nicht ganz korrekt, zum Beispiel $g(n) = \log(n)$ oder $g(n) = \sqrt{n}$.

Trotzdem verwenden wir die Landau-Symbole $\mathcal{O}$ oder Ω auch für solche Funktionen. In diesen Fällen sind die entsprechenden nicht-negativen, ganzzahligen Funktionen $\hat{f}(n) = \max\{\lceil f(n) \rceil, 0\}$ gemeint.

Dabei bezeichnet $\lceil x \rceil$ die kleinste ganze Zahl, die nicht kleiner als x ist.

Wichtige Aufwandsklassen:

$\mathcal{O}(1)$	konstant
$\mathcal{O}(\log(n))$	logarithmisch
$\mathcal{O}(\log^k(n))$	poly-logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \cdot \log(n))$	
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(n^k)$	polynomiell
$\mathcal{O}(2^n)$	exponentiell

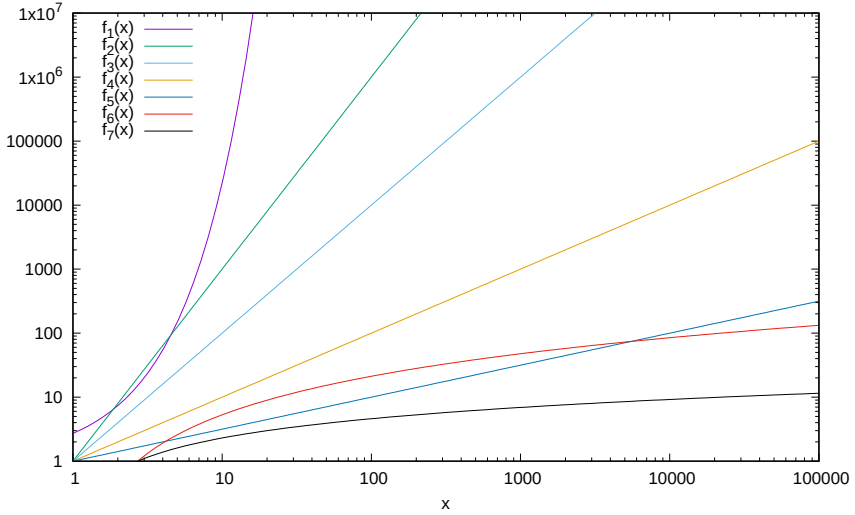
Es gelten folgende Inklusionen.

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\log^2(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \cdot \log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n)$$

Übung 1. *Warum geben wir bei der logarithmischen Laufzeit keine Basis des Logarithmus an?*

Aufwandsklassen

$f_1 : e^x$
$f_2 : x^3$
$f_3 : x^2$
$f_4 : x$
$f_5 : \sqrt{x}$
$f_6 : \ln^2(x)$
$f_7 : \ln(x)$



Aufwandsklassen

Weil „wir wissen, was gemeint ist“, schreiben wir manchmal seltsame Dinge[(1)]:

- $3n^2 + 4n \;=\; 3n^2 + \mathcal{O}(n) \;=\; \mathcal{O}(n^2)$
  - Damit wollen wir aussagen, dass das Wachstum der Funktion $3n^2 + 4n$ im wesentlichen durch den Term $3n^2$ bestimmt wird und der vernachlässigte Term nur linear wächst.
  - Man liest eine solche „Gleichung“ also von links nach rechts, wobei die „Genauigkeit“ von links nach rechts abnimmt.
- $T(n) = 2 \cdot T(n/2) + \Theta(n)$
  - Diese „Gleichung“ beschreibt die Laufzeit eines rekursiven Algorithmus.
  - Es gibt zwei rekursive Aufrufe mit jeweils einer halb so großen Eingabe.
  - Auf jeder Rekursionsebene hat der iterative Teil des Algorithmus einen linearen Aufwand.

[(1)] Versuchen Sie das einmal „sauber“ aufzuschreiben.

Oft benötigen wir asymptotische Aufwandsabschätzungen für mehrstellige Funktionen:

- naive Textsuche: verschiebe ein Muster der Länge m über einen Text der Länge n und prüfe stellenweise auf Gleichheit. Dabei ergibt sich als Laufzeit $(n - (m - 1)) \cdot m$.
- Bei der Tiefensuche auf einem Graphen mit n Knoten und m Kanten ergibt sich eine Laufzeit, die von n und m abhängt.

Wir erweitern daher die Groß-O-Notation auf zweistellige Funktionen $g : \mathbb{N}_0^2 \to \mathbb{N}_0$.

$$\mathcal{O}(g) := \left\{ f : \mathbb{N}_0^2 \to \mathbb{N}_0 \;\middle|\; \begin{array}{l} \exists c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall x, y \in \mathbb{N} : \\ x, y \geq n_0 \;\Rightarrow\; f(x, y) \leq c \cdot g(x, y) \end{array} \right\}$$

Worst-Case Komplexität

Definition: (Worst-Case Komplexität)
W_n: Menge der zulässigen Eingaben der Länge n.
$A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

Worst-Case Komplexität (im schlechtesten Fall):

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

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

Average-Case Komplexität

Definition: (Average-Case Komplexität)
W_n: Menge der zulässigen Eingaben der Länge n.
$A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

Average-Case Komplexität (erwarteter Aufwand):

$$\overline{T}_A(n) = \frac{1}{|W_n|} \cdot \sum_{w \in W_n} A(w)$$

ist die mittlere Anzahl von Schritten, die Algorithmus A benötigt, um eine Eingabe der Größe n zu bearbeiten.

Wir setzen hier und im Verlauf dieser Veranstaltung eine Gleichverteilung voraus.
$\longrightarrow$ arithmetischer Mittelwert

Fragen / Probleme:

- Worüber bildet man den Durchschnitt?
- Sind alle Eingaben der Länge N gleich wahrscheinlich? oder: Binomial-, Normal-, Poissonverteilung?
- Technisch oft sehr viel schwieriger durchzuführen als die worst-case Analyse.

Murphys Gesetz: Alles was schiefgehen kann, wird auch schiefgehen! Für uns heißt das: Immer wenn ich das Programm ausführe, warte ich ewig.

Average-Case Untersuchungen sind ungeeignet für kritische Anwendungen, bei denen maximale Reaktionszeiten garantiert werden müssen! → Vorlesung *Echtzeitsysteme*

Warum worst-case Komplexität?

- Obere Schranke der Laufzeit: Wir können uns sicher sein, dass das Programm nach einer gewissen Zeit fertig ist.
- Der schlimmste Fall kommt bei manchen Algorithmen oft vor, z.B. beim Suchen nach Daten, die in einer Datenbank nicht vorhanden sind.
- Der durchschnittliche Fall ist oft fast genauso schlecht wie der schlimmste Fall.

Übersicht

Definition: Ein *Alphabet* ist eine endliche, nichtleere Menge von Zeichen (Symbole oder Buchstaben genannt).

Definition: Endliche Folgen $(x_1, \ldots, x_k)$ mit $x_i \in A$ heißen *Wörter* der Länge k über dem Alphabet A.

Definition: Die Menge aller Wörter über einem Alphabet A wird mit A^* bezeichnet. Das leere Wort wird durch das Symbol ϵ dargestellt.

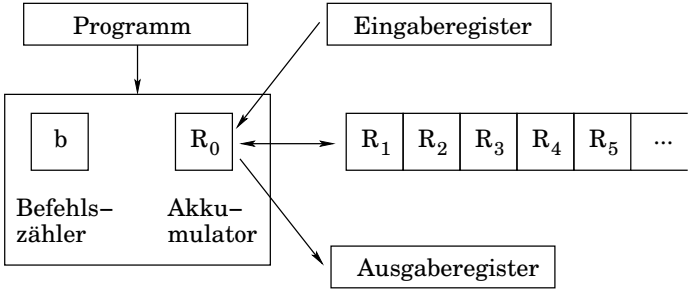
Eine Teilmenge von A^* wird als *Sprache* bezeichnet.

Satz: Die Menge A^* aller Wörter über einem Alphabet A ist abzählbar.

Übung 2. *Beweisen Sie den letzten Satz.*

Random Access Machine (RAM) (Registermaschine)

- fester Befehlssatz: read, write, add, sub, goto, if goto, ...
- abzählbar unendlich viele Speicherzellen $R_0, R_1, R_2, \ldots$
- spezielles Register R_0 für Arithmetik $\rightarrow$ Akkumulator
- spezielles Register für Eingabe
- spezielles Register für Ausgabe



einige Befehle:

- *READ:* $R_0 := head(e)$, $e := tail(e)$, $b := b + 1$
- *WRITE:* $a := a \oplus R_0$, $b := b + 1$
- *ADD R_i:* $R_0 := R_0 + R_i$, $b := b + 1$
- *ADD i:* $R_0 := R_0 + i$, $b := b + 1$
- *LOAD R_i:* $R_0 := R_i$, $b := b + 1$
- *ILOAD R_i:* $R_0 := R_{R_i}$, $b := b + 1$
- *ACCEPT* und *REJECT* zum Vergleich mit Turingmaschinen

Kostenmaße:

- *uniform:* Anzahl der ausgeführten Befehle bzw. der benutzten Speicherzellen.
- *logarithmisch:* Berücksichtige die binäre Länge der benutzten Operanden bei den jeweiligen Befehlen.

Beispiel: ggT(x, y) berechnen, zu Beginn $R_1 = x$, $R_2 = y$

Zeile	Anweisung	Kommentar
1:	LOAD (ACC FROM) R_1	$R_0 := R_1$
2:	STORE (ACC TO) R_3	$R_3 := R_0$
3:	SUB R_2 (FROM ACC)	$R_0 := R_0 - R_2$
4:	JLZ 6	jump to Z6 if $R_0 < 0$
5:	STORE (ACC TO) R_3	$R_3 := R_0$
6:	LOAD (ACC FROM) R_2	$R_0 := R_2$
7:	STORE (ACC TO) R_1	$R_1 := R_0$
8:	LOAD (ACC FROM) R_3	$R_0 := R_3$
9:	STORE (ACC TO) R_2	$R_2 := R_0$
10:	JNEZ 1	jump to Z1 if $R_0 \neq 0$
11:	HALT	

Am Ende steht das Ergebnis in R_1.

Veranschaulichung: Zu Beginn gelte $R_1 = x$ und $R_2 = y$.

1.	$R_0 := R_1$	$R_3 := x$
2.	$R_3 := R_0$	
3.	$R_0 := R_0 - R_2$	$Z6$ falls $x - y < 0$
4.	$Z6$ if $R_0 < 0$	
5.	$R_3 := R_0$	$\rightarrow R_3 := x - y$
6.	$R_0 := R_2$	$R_1 := y$
7.	$R_1 := R_0$	
8.	$R_0 := R_3$	$R_2 := R_3$
9.	$R_2 := R_0$	
10.	$Z1$ if $R_0 \neq 0$	$\rightarrow$ Z1, falls $x \neq y$

als Pseudo-Code:

repeat
  $t := x$
  if $x \geq y$ **then**
    $t := x - y$
  $x := y$
  $y := t$
until $t = 0$

Sobald Zeile 9 abgearbeitet ist, gilt

- ggT(x,y) = ggT(y, $x - y$), falls $x \geq y$ ist,
- andernfalls gilt ggT(x,y) = ggT(y, x).

Feststellung: Die RAM ähnelt unseren heutigen Computern und entspricht intuitiv unserem Berechenbarkeitsbegriff:

> *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$ *ist berechenbar, falls es ein RAM-Programm* p *gibt, so dass gilt:* p *berechnet* m *zur Eingabe* $n_1, \ldots, n_k$ *genau dann wenn* $f(n_1, \ldots, n_k) = m$.

Wenn wir sagen, eine RAM berechnet m, dann meinen wir damit, dass m ins Ausgaberegister geschrieben wird und die Maschine hält. Jede Berechnung ist also endlich.

Anmerkung: Anstelle eines RAM-Programms können wir auch ein Programm in einer höheren Programmiersprache wie C/C++, Java oder Python verwenden, da diese Programme durch einen Compiler in Assembler-Code umgesetzt werden.

Die folgenden Beispiele sind dem sehr empfehlenswerten Buch von Uwe Schöning entnommen: Theoretische Informatik – kurzgefasst.

Übung 3. *Ist folgende Funktion berechenbar?*

$$f(n) = \begin{cases} 1, & \text{falls die Dezimalbruchentwicklung von } \pi \text{ mit } n \text{ beginnt} \\ 0, & \text{sonst} \end{cases}$$

Beispiele: $f(314) = 1$, $f(31417) = 0$, $f(31415) = 1$

Übung 4. *Ist folgende Funktion berechenbar?*

$$g(n) = \begin{cases} 1, & \text{in der Dezimalbruchentwicklung von } \pi \text{ kommt irgendwo } n \text{ vor} \\ 0, & \text{sonst} \end{cases}$$

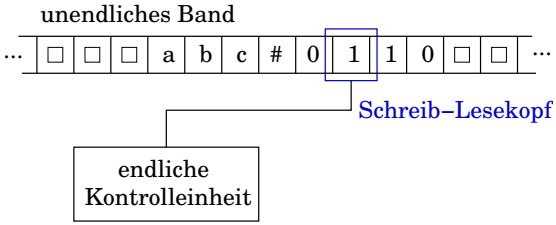
$\pi = 3.14159265358979323846264338327950288...$

Beispiele: $g(314) = 1$, $g(6535) = 1$, $g(358979) = 1$

Übung 5. *Ist folgende Funktion berechenbar?*

$$h(n) = \begin{cases} 1, & \text{die Dezimalbruchentwicklung von } \pi \\ & \quad \text{enthält irgendwo } n\text{-mal hintereinander eine 7} \\ 0, & \text{sonst} \end{cases}$$

deterministische Turing-Maschine (dTM)



- Das potentiell unendliche Band ist in Felder unterteilt.
- Jedes Feld kann ein einzelnes Zeichen des Arbeitsalphabets der Maschine enthalten.
- Auf dem Band kann sich der Schreib-Lesekopf bewegen.
- Nur das Zeichen, auf dem sich dieser Kopf gerade befindet, kann im momentanen Rechenschritt verändert werden.
- Der Kopf kann in einem Rechenschritt nur um maximal eine Position nach links oder rechts bewegt werden.
- Noch nicht besuchte Felder enthalten das Blank-Symbol □.

Formal: Eine Turingmaschine M ist gegeben durch ein 7-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$.

- Z ist die Zustandsmenge.
- Σ ist das Eingabealphabet.
- Γ ist das Ausgabealphabet mit $\Sigma \subseteq \Gamma$.
- $\delta : Z \times \Gamma \to Z \times \Gamma \times \{L, N, R\}$ ist die partielle Übergangsfunktion, die (u.a.) für keinen Wert aus $E \times \Gamma$ definiert ist.
- $z_0 \in Z$ ist der Startzustand.
- $\square$ ist das Blank-Symbol.
- $E \subsetneq Z$ ist die Menge der akzeptierenden Endzustände.

Informal bedeutet $\delta(z, a) = (z', b, x)$ folgendes:

Wenn sich Maschine M

- im Zustand z befindet und
- unter dem Schreib-Lesekopf das Zeichen a steht,

dann

- geht M im nächsten Schritt in den Zustand z' über,
- schreibt auf den Platz von a das Zeichen b auf das Band
- und führt danach die Kopfbewegung $x \in \{L, N, R\}$ aus mit L für links, N für neutral (stehenbleiben) und R für rechts.

Dabei ist $z = z'$ und $a = b$ möglich.

Definition: Eine Konfiguration $\alpha z \beta$ ist eine *Momentaufnahme* der Turingmaschine.

- $\alpha\beta \in \Gamma^*$ ist der nicht-leere, schon besuchte Teil des Bandes.
- Der Schreib-Lesekopf steht auf dem ersten Zeichen von β.
- z ist der Zustand, in dem sich die Maschine befindet.

Definition: Startkonfiguration $z_0 x$

- Die Eingabe $x \in \Sigma^*$ steht schon auf dem Band.
- Der Schreib-Lesekopf steht auf dem ersten Zeichen der Eingabe x.
- Die Maschine ist im Startzustand z_0.

Übung 6. *Geben Sie eine Turingmaschine an, die auf eine Eingabe, interpretiert als Binärzahl, eine 1 hinzu addiert.*

Definition: Die von einer Turingmaschine M *akzeptierte Sprache* ist wie folgt definiert:

$$T(M) = \{x \in \Sigma^* \mid z_0 x \overset{*}{\longmapsto} \alpha z \beta; \alpha, \beta \in \Gamma^*; z \in E\}$$

Dabei bedeutet $z_0 x \overset{*}{\longmapsto} \alpha z \beta$, dass die Maschine ausgehend von der Startkonfiguration $z_0 x$ nach endlich vielen Schritten eine Konfiguration $\alpha z \beta$ erreicht und hält.

Beachte: Für ein $x \notin T(M)$ darf die Maschine M in eine Endlosschleife gehen!

Definition: $L \subseteq \Sigma^*$ heißt *rekursiv aufzählbar*, wenn es eine dTM gibt, die L akzeptiert.

Definition: Falls M die Sprache L akzeptiert und für alle $x \in \Sigma^*$ hält, so *entscheidet* M die Sprache L.

$L \subseteq \Sigma^*$ heißt *entscheidbar* oder auch *rekursiv*, wenn es eine dTM gibt, die L entscheidet.

Übung 7. *Geben Sie eine Turingmaschine an, die* $L = \{0^n 1^n \mid n \geq 1\}$ *entscheidet.*

Bei der RAM haben wir von *Berechnung einer Funktion* gesprochen, bei der dTM sprechen wir von *akzeptierten Sprachen*. Wo ist da der Zusammenhang?

Eine dTM $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ berechnet eine Funktion $f : \Sigma^* \to \Sigma^*$ genau dann, wenn für ein $z_e \in E$ gilt:

$$z_0 x \overset{*}{\vdash\!\!\!-\!\!\!-} z_e y \iff f(x) = y$$

Die Turingmaschine schreibt also y auf das Band und geht in einen akzeptierenden Endzustand.

Für Funktionen $f : \mathbb{N}^r \to \mathbb{N}^s$, die auf natürlichen Zahlen definiert sind, kann man entsprechend definieren: Die dTM M berechnet f genau dann, wenn M gestartet mit $x = 0^{n_1+1}10^{n_2+1}1 \ldots 10^{n_r+1}$ den Wert $y = 0^{m_1+1}10^{m_2+1}1 \ldots 10^{m_s+1}$ auf das Band schreibt[(2)], falls $f(n_1, \ldots, n_r) = (m_1, \ldots, m_s)$ ist.

[(2)] Hier werden die Zahlenwerte unär kodiert, damit man $\Sigma = \{0, 1\}$ wählen kann. Man könnte auch eine binäre Kodierung der Zahlen nutzen und ein Symbol wie # verwenden, um die Zahlen voneinander zu trennen, dann wäre $\Sigma = \{0, 1, \#\}$.

Das *Akzeptieren von Sprachen* und das *Berechnen von Funktionen* ist sehr ähnlich. Sei M eine dTM, die die Funktion $f : \Sigma^* \to \Sigma^*$ berechnet. Dann kann sehr einfach eine dTM M' konstruiert werden, die die Sprache $L = \{(x, f(x))\}$ akzeptiert und dafür M nutzt.

Beispiel: Eine dTM M berechne die Funktion $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit $f(x, y) = x + y$. Die zur Funktion f entsprechende Sprache ist

$$L = \{(x, y, z) \mid x + y = z\}$$

Eine dTM M', die die Sprache L akzeptiert, nutzt die dTM M wie folgt:

- M' verhält sich wie M gestartet mit $0^{x+1}10^{y+1}$.
- M' akzeptiert genau dann, wenn das Ergebnis gleich 0^{z+1} ist.

Die umgekehrte Richtung ist etwas problematischer. Sei M eine dTM, die die Sprache L akzeptiert. Dann kann eine dTM M' konstruiert werden, die die *charakteristische Funktion* χ_L berechnet:

$$\chi_L(x) = 1 \iff x \in L$$

Die dTM M' verhält sich wie M gestartet mit x.

- Wenn M die Eingabe x akzeptiert, löscht M' das Band und schreibt eine 1.
- Wenn M die Eingabe x nicht akzeptiert, löscht M' das Band und schreibt eine 0.

Problem: Wenn die Maschine M die Eingabe x nicht akzeptiert, kann es sein, dass M nicht hält. In diesem Fall kann M' das Band nicht löschen und eine 0 schreiben.

Wenn M die Sprache L entscheidet, also insbesondere für jede Eingabe hält, so kann die charakteristische Funktion χ_L berechnet werden.

Im weiteren wollen wir die Begriffe Akzeptanz und Entscheidung nutzen.

Satz: Die Klasse der entscheidbaren Sprachen ist abgeschlossen gegen Komplementbildung.

Beweisidee: Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ eine dTM, die L entscheidet.

Folgende dTM $\bar{M}$ entscheidet $\bar{L}$:

$$\bar{M} = (Q \cup \{q^*\}, \Sigma, \Gamma, \bar{\delta}, q_0, \square, \{q^*\})$$

- Die akzeptierenden Endzustände von M sind bei $\bar{M}$ nicht akzeptierend.
- Stattdessen gibt es bei $\bar{M}$ einen neuen akzeptierenden Endzustand q^*.
- $\bar{\delta}$ ist definiert wie δ, enthält aber Zusatzregeln: Falls M im Zustand $q \notin F$ hält, geht $\bar{M}$ erst in den Zustand q^* und hält dann.

Satz: Eine Sprache L ist entscheidbar, genau dann wenn L und $\bar{L}$ rekursiv aufzählbar sind.

Beweisidee:

$\Rightarrow$ Da L entscheidbar ist, ist auch $\bar{L}$ entscheidbar.

Außerdem gilt: Jede entscheidbare Sprache ist insbesondere rekursiv aufzählbar.

$\Leftarrow$ Seien M_1 und M_2 die TMs, die L bzw. $\bar{L}$ akzeptieren.

Die dTM M führt jeweils einen Schritt von M_1 und M_2 aus und hält, falls eine der beiden Maschinen hält. M akzeptiert, falls M_1 akzeptiert, und verwirft, falls M_2 akzeptiert.

Kostenmaße:

- *Laufzeit:* Die Anzahl ausgeführter Konfigurationsübergänge.
- *Speicherplatz:* Die maximale Länge einer Konfiguration während einer Rechnung.

Programmiertechniken für Turingmaschinen:

- Im Zustand merken.
- Nutzen mehrerer Spuren.
- Nutzen mehrerer Bänder.
- Unterprogramme.

Im Zustand merken: Man kann eine von endlich vielen Informationen aus einer endlichen Menge A speichern, indem man sie sich „im Zustand merkt“. Dazu ersetzt man Q durch $Q \times A$.

Beispiel: Wir wollen testen, ob bei Eingabe $x_1 \dots x_n \in \Sigma^+$ der Buchstabe x_1 in $x_2 \dots x_n$ vorkommt. Wir merken uns x_1 im Zustand.

Sei $\Gamma = \Sigma \cup \{\square\}$, $Q = (\{q_0\} \times \Sigma) \cup \{q_0, q_1\}$, Startzustand q_0, $F = \{q_1\}$.

$$\begin{array}{lcll} \delta(q_0, a) & = & ([q_0, a], a, R) & \forall a \in \Sigma \\ \delta([q_0, a], a) & = & (q_1, a, N) & \forall a \in \Sigma \\ \delta([q_0, a], b) & = & ([q_0, a], b, R) & \forall a, b \in \Sigma, a \neq b \end{array}$$

Nutzen mehrerer Spuren: Um auf k Spuren unterschiedliche Worte schreiben zu können, benutzen wir als Bandalphabet Γ^k anstatt Γ.

E	R	I	K
M	A	G	
L	E	N	A
	↑		

Im obigen Beispiel ist $k = 3$, der Bandinhalt der Kopfposition ist $[\mathrm{RAE}] \in \{A, \ldots, Z\}^3$.

Anwendung dieser Technik im Beweis zu folgendem Satz:

Satz: Jede $t(n)$-zeit-, $s(n)$-platzbeschränkte k-Band-dTM kann durch eine 1-Band-dTM in Zeit $O(t(n) \cdot s(n))$ auf Platz $O(s(n))$ simuliert werden.

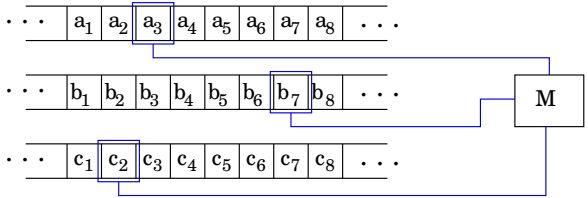
k-Band-dTM: Eine k-Band Turingmaschine hat nicht nur ein Band und einen Kopf, sondern k Bänder mit je einem Kopf.

- Die Übergangsfunktion ist von der Form $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, N\}^k$.
- Zu Beginn steht die Eingabe auf Band 1, sonst stehen überall Blanks.
- Die Arbeitsweise ist analog zu 1-Band-dTMs definiert.

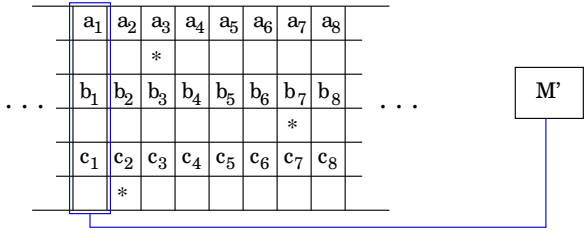
Mehrband-dTM verwenden wir um Unterprogramme zu realisieren (siehe dazu nächsten Abschnitt).

Übung 8. *Geben Sie jeweils eine 1-Band- und eine 2-Band-dTM an, die die Sprache* $\{ww^R \mid w \in \{0,1\}^*\}$ *entscheidet.*

Beweis: Sei k die Anzahl der Bänder und Γ das Arbeitsalphabet von M. Wir unterteilen das Band von M' in $2k$ Spuren, so dass eine Konfiguration von M



simuliert wird durch:



Als Arbeitsalphabet von M' wählen wir $\Gamma' = \Gamma \cup (\Gamma \cup \{\star\})^{2k}$.

M' simuliert einen Schritt von M wie folgt:

- Der Kopf von M' steht zu Beginn des Simulationsschrittes links von allen $\star$-Marken.
- M' durchläuft das Band bis alle $\star$-Marken überschritten wurden und merkt sich die von M' gelesenen Zeichen an den $\star$-Positionen im Zustand. Dazu wird nur ein endlicher Speicher benötigt, also $Q' = Q \times (\Gamma^k \times Q))$.
- Nun weiß M', welche Zeile der δ-Funktion von M anzuwenden ist.
- M' läuft wieder nach links über alle $\star$-Marken hinweg und führt alle entsprechenden Änderungen aus (Änderung der Inschriften auf ungeraden Spuren und Versetzen der $\star$-Marken) und merkt sich den neuen Zustand von M im Zustand.

Unterprogramme: - hier nur für Mehrband-dTM -

- Eine Teilmenge von Zuständen, die für das UP reserviert sind.
- Das UP wird von einem bestimmten Zustand der dTM aufgerufen.
- Die zu übergebenden Daten werden auf ein für UPe reserviertes Band der dTM kopiert.
- Das UP arbeitet nur auf diesem Band und das Ergebnis der Berechnung wird am Ende des UPs auf andere Bänder der dTM kopiert, um für weitere Berechnungen zur Verfügung zu stehen.
- Die Maschine geht nach Beendigung des UPs in einen Zustand über, der nicht zur Menge der für das UP reservierten Zustände gehört.

Beispiel: In der Turingmaschine für die Sprache $L = \{0^n 1^n \mid n \geq 1\}$ können die Zustände z_0, z_1 als UP aufgefasst werden.

Satz: dTMs und RAMs können mit polynomiellem Zeitverlust gegenseitig simuliert werden.

Beweisidee: Schalte mehrere Turingmaschinen hintereinander.

$M_1 = (Z_1, \Sigma, \Gamma_1, \delta_1, z_1, \square, E_1)$ und $M_2 = (Z_2, \Sigma, \Gamma_2, \delta_2, z_2, \square, E_2)$ seien zwei Turingmaschinen. Dann bezeichnet

$$start \longrightarrow M_1 \longrightarrow M_2 \longrightarrow stop$$

eine neue TM $M = (Z_1 \cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, \square, E_2)$ wobei

- o.B.d.A. $Z_1 \cap Z_2 = \emptyset$ und
- $\delta = \delta_1 \cup \delta_2 \cup \{(z_e, a, z_2, a, N) \mid z_e \in E_1, a \in \Gamma_1\}$

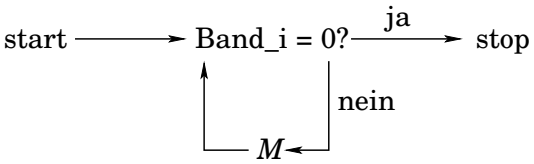
Beweisidee: (Fortsetzung)

- Bezeichne `Band := Band + 1` unsere Addiermaschine.
- Daraus bauen wir eine k-Band TM `Band_i := Band_i + 1`, bei der die Aktionen nur auf Band `i` ablaufen und alle anderen Bänder unverändert bleiben.
- In ähnlicher Weise können wir Maschinen konstruieren für:
  - `Band_i := Band_i - 1`
  - `Band_i := 0`
  - `Band_i := Band_j`
  - `Band_i = 0?`
- Wir konstruieren eine TM, die vom Endzustand z_{e_1} von M nach M_1 übergeht und von z_{e_2} aus nach M_2 (Verzweigung):

$$start \longrightarrow M \begin{array}{ll} \nearrow^{z_{e_1}} & M_1 \longrightarrow stop \\ \searrow_{z_{e_2}} & M_2 \longrightarrow stop \end{array}$$

Beweisidee: (Fortsetzung)

- Wir können auch Schleifen realisieren:



Man erkennt, dass wir einfache Programmiersprachen-ähnliche Konzepte mit einer Mehrband-TM simulieren können:

- Die Bandinhalte können als Variablenwerte angesehen werden,
- es gibt einfache Wertzuweisungen,
- die Hintereinanderausführung von Programmen ist möglich,
- Verzweigungen und Schleifen können programmiert werden.

Churche These (1936): Die im intuitiven Sinne berechenbaren Funktionen sind genau die, die durch Turingmaschinen berechenbar sind.

- Church: λ-Kalkül
- Kleene: μ-Rekursion
- Markov: Markov-Algorithmen

Alle vorgeschlagenen Formalisierungen von Berechenbarkeit sind äquivalent zur Berechenbarkeit durch eine Turingmaschine.

nichtdeterministische Turing-Maschine (nTM)

- Eine nTM besitzt anstelle der Übergangsfunktion eine Übergangsrelation, d.h. jede Konfiguration hat mehrere mögliche Nachfolgekonfigurationen.

  Daher sind bei einer festgelegten Maschine und einer festgewählten Eingabe mehrere Resultate möglich.
- Eine nTM akzeptiert eine Eingabe gdw. *es existiert eine* Berechnung von der Startkonfiguration in eine akzeptierende Endkonfiguration.

Übung 9. *Geben Sie eine (2-Band) nTM an,*

- *die alle geradlängigen Palindrome über dem Alphabet* $\{0,1\}$ *erkennt, also die Sprache* $L = \{ww^R \mid w \in \{0,1\}^*\}$ *akzeptiert.*
- *die die Sprache* $L = \{ww \mid w \in \{0,1\}^*\}$ *akzeptiert.*

Der folgende Satz zeigt, dass der Berechenbarkeitsbegriff durch den Nichtdeterminismus nicht erweitert wird!

Satz: Eine polynomiell zeitbeschränkte nTM kann durch eine exponentiell zeitbeschränkte dTM simuliert werden.

Übung 10. *Beweisen Sie den letzten Satz.*

zur Erinnerung:

- Für eine Menge M ist $\mathcal{P}(M) := \{A \mid A \subseteq M\}$ die Potenzmenge von M.
- Beispiel: Für $M = \{1, 2, 3\}$ ergibt sich $\mathcal{P}(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
- Für eine natürliche Zahl n bezeichnet $[n]$ die Menge $\{1, 2, 3, \ldots, n\}$.

Satz: Es gibt Funktionen $f : \{0,1\}^* \to \{0,1\}$, die nicht durch ein C-Programm berechenbar sind.

Beweisidee:

- Die Menge P der C-Programme ist abzählbar unendlich: endlicher Text über einem endlichen Alphabet A mit Syntax-Check durch Compiler.
- Jedes Programm berechnet genau eine Funktion:
  $f : A^* \to A^* \cup \{\text{Programm hält nicht}\}$
- P ist abzählbar, die Menge $F = \{f : \{0,1\}^* \to \{0,1\}\}$ ist überabzählbar.

$\Rightarrow$ Es existiert ein $f \in F$ das nicht von einem Programm aus P berechnet wird.

Frage: Warum ist F überabzählbar?

Bisher: Special Purpose Machines lösen nur genau ein Problem.

Wie können wir programmierbare (universelle) Computer darstellen? Das Programm soll dabei eine Beschreibung einer special purpose machine sein.

Gödelnummer: Sei M eine 1-Band-dTM mit $Q = \{q_1, \ldots, q_m\}$, $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \square\}$.

Sei $X_1 \hat{=} 0$, $X_2 \hat{=} 1$, $X_3 \hat{=} \square$, $D_1 \hat{=} L$, $D_2 \hat{=} R$, $D_3 \hat{=} N$.

Codiere $\delta(q_i, X_j) = (q_k, X_l, D_m)$ durch $0^i 1 0^j 1 0^k 1 0^l 1 0^m$.

δ ist eine endliche Funktionstabelle, daher sei $Code_t$ die Codierung der t-ten Zeile, $1 \leq t \leq g$. Die *Gödelnummer* von M ist

$$\langle M \rangle = 111 Code_1 11 Code_2 11 Code_3 \ldots 11 Code_g 111$$

wobei $g \leq m \cdot |\Gamma|$ die Größe von δ ist.

Definition: Eine TM M_0 heißt *universell*, falls für jede 1-Band-dTM M und jedes $x \in \{0,1\}^*$ gilt:

- M_0 gestartet mit $\langle M \rangle x$ hält genau dann, wenn M gestartet mit x hält.
- Falls M gestartet mit x hält, berechnet M_0 gestartet mit $\langle M \rangle x$ den gleichen Output wie M gestartet mit x.
- Insbesondere akzeptiert M_0 den Input $\langle M \rangle x$ genau dann, wenn M den Input x akzeptiert.

Satz: Es gibt eine universelle 2-Band-dTM, die jede $t(n)$-zeit- und $s(n)$-platzbeschränkte 1-Band-dTM in Zeit $O(t(n))$ auf Platz $O(s(n))$ simuliert.

Die Länge des Programms, also die Länge von $\langle M \rangle$, wird in der Groß-O-Notation als Konstante angenommen.

Beweisidee Teil 1: Codierung einer Konfiguration

Eine Konfiguration $\alpha q_i X_j \beta$ von M wird von M_0 wie folgt codiert:

- Auf Band 1 steht $\langle M \rangle$ und der Zustand q_i, codiert durch 0^i.
- Auf Band 2 steht die Bandinschrift $\alpha X_j \beta$ von M.
- Der Kopf von Band 2 steht auf X_j.

Zu Beginn gilt:

- Auf Band 2 steht die Eingabe x, der Kopf befindet sich auf dem ersten Zeichen von x.
- Der Startzustand ist auf Band 1 notiert.

Am Ende gilt:

- Halte, falls keine Nachfolgekonfiguration existiert.
- Akzeptiere, falls der akzeptierende Endzustand von M auf Band 1 hinter $\langle M \rangle$ steht.

Beweisidee Teil 2: Simulation eines Schrittes

Die TM M befinde sich in der Konfiguration $\alpha q_i X_j \beta$:

- Suche auf Band 1 in $\langle M \rangle$ die Zeichenreihe 110^i10^j, also den Anfang der Codierung des Übergangs $\delta(q_i, X_j)$.
  - $j \in \{1, 2, 3\}$ kann im Zustand gespeichert werden.
  - 0^i wird durch Vergleich mit der Codierung von $q_i \hat{=} 0^i$ auf Band 1 gefunden. (Warum q_i nicht im Zustand speichern?)
- Lese die dahinterstehende Zeichenreihe der Form $10^k10^l10^m$.
  - Ersetze die Codierung 0^i von q_i auf Band 1 durch 0^k, also durch die Codierung des neuen Zustands q_k.
  - Speichere die Information l und m über den auszuführenden nächsten Schritt im Zustand: „Überschreibe die Zelle der Kopfposition mit X_l und bewege den Kopf gemäß D_m“.
- Verändere Band 2 entsprechend dem im Zustand gespeicherten Befehl.

Definition: Die *Diagonalsprache* $DIAG$ ist wie folgt definiert:

$$DIAG = \{\langle M \rangle \mid M \text{ ist dTM, die } \langle M \rangle \text{ nicht akzeptiert}\}$$

Satz: $DIAG$ ist nicht rekursiv aufzählbar.

Beweis durch Widerspruch.
Angenommen, die dTM $\bar{M}$ akzeptiert $DIAG$, also: $\bar{M}$ akzeptiert x genau dann wenn $x \in DIAG$. Was macht $\bar{M}$ mit Eingabe $\langle \bar{M} \rangle$?

- Fall 1:
  - $\langle \bar{M} \rangle \in DIAG \Rightarrow \bar{M}$ akzeptiert die Eingabe $\langle \bar{M} \rangle$ nicht.
  - $\bar{M}$ akzeptiert $DIAG \Rightarrow \bar{M}$ akzeptiert $\langle \bar{M} \rangle$, wenn $\langle \bar{M} \rangle \in DIAG$; ein Widerspruch.
- Fall 2:
  - $\langle \bar{M} \rangle \notin DIAG \Rightarrow \bar{M}$ akzeptiert die Eingabe $\langle \bar{M} \rangle$.
  - $\bar{M}$ akzeptiert $DIAG \Rightarrow \bar{M}$ akzeptiert $\langle \bar{M} \rangle$ nicht, wenn $\langle \bar{M} \rangle \notin DIAG$; ein Widerspruch.

Explizite Darstellung des Beweises als Diagonalisierung

Sei $M_1, M_2, \ldots$ die Folge aller in der Reihenfolge ihrer Gödelnummern aufgezählten dTMs.

Betrachte die unendliche Matrix, deren Zeilen mit $\langle M_1 \rangle, \langle M_2 \rangle, \ldots$, und deren Spalten mit $M_1, M_2, \ldots$ nummeriert sind.

An der Position M_i, $\langle M_j \rangle$ wird eingetragen, ob M_i die Eingabe $\langle M_j \rangle$ akzeptiert (Eintrag „a“) oder nicht akzeptiert (Eintrag „na“).

	M_1	M_2	M_3	$\cdots$
$\langle M_1 \rangle$	*a*	*na*	*a*	$\cdots$
$\langle M_2 \rangle$	*na*	*na*	*a*	$\cdots$
$\langle M_3 \rangle$	*na*	*na*	*a*	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Die Spalte unter M_i heißt *Akzeptanz-Folge* von M_i.

Wir nehmen wieder die Existenz einer dTM $\bar{M}$ an, die *DIAG* akzeptiert.

	M_1	M_2	M_3	$\cdots$
$\langle M_1 \rangle$	a	na	a	$\cdots$
$\langle M_2 \rangle$	na	na	a	$\cdots$
$\langle M_3 \rangle$	na	na	a	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$\Rightarrow$

DIAG	M_1	M_2	M_3	$\cdots$
$\langle M_1 \rangle$	na	na	a	$\cdots$
$\langle M_2 \rangle$	na	a	a	$\cdots$
$\langle M_3 \rangle$	na	na	na	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Wie sieht die Akzeptanz-Folge von $\bar{M}$ aus?

- Definition von *DIAG*: An Stelle i hat sie den Eintrag a falls M_i die Eingabe $\langle M_i \rangle$ nicht akzeptiert, sonst den Eintrag na.
- Somit ist $\bar{M}$ keine der dTMs $M_1, M_2, \ldots$, da sich die Akzeptanz-Folge von M_i an Stelle i von der von $\bar{M}$ unterscheidet.
- Somit kann es $\bar{M}$ nicht geben, da wir ja in $M_1, M_2, \ldots$ alle dTMs aufgezählt haben. Widerspruch!

Halteproblem: Stoppt ein Programm zu einer bestimmten Eingabe?

$$H = \{\langle M \rangle x \mid M \text{ ist dTM, die gestartet mit Eingabe } x \text{ hält}\}$$

Halteproblem bei leerem Band:

$$H_0 = \{\langle M \rangle \mid M \text{ ist dTM, die gestartet mit Input } \epsilon \text{ hält}\}$$

Satz: H und H_0 sind rekursiv aufzählbar.

- H ist rekursiv aufzählbar, da die universelle dTM H akzeptiert, falls wir sie so modifizieren, dass sie immer akzeptiert, falls sie hält.
- H_0 ist daher natürlich auch rekursiv aufzählbar.

Um zu zeigen, dass H und H_0 nicht entscheidbar sind, nutzen wir keinen Diagonalisierungsbeweis, sondern eine andere Methode, die anwendet, dass $DIAG$ nicht entscheidbar ist.

Vorüberlegung: $\overline{DIAG}$ ist nicht entscheidbar, da die Klasse der entscheidbaren Sprachen abgeschlossen ist gegenüber Komplementbildung.

$$\overline{DIAG} = \{w \in \{0,1\}^* \mid \begin{array}{l} w \text{ ist keine Codierung einer TM, oder} \\ w = \langle M \rangle \text{ und } M \text{ akzeptiert } \langle M \rangle\} \end{array}$$

$\overline{DIAG}$ nennt man auch *Selbstanwendbarkeitsproblem* oder *spezielles Halteproblem*.

Reduzierbarkeit

Definition: Eine Sprache L heißt reduzierbar auf L', falls es eine totale berechenbare Funktion $f : \Sigma^* \longrightarrow \Sigma^*$ gibt mit $x \in L \iff f(x) \in L'$.

- Wir schreiben dann: $L \leq L'$ (mittels f)
- Idee: L ist nicht schwieriger zu lösen als L'.

Satz: Falls L nicht entscheidbar ist und $L \leq L'$ (mittels f) gilt, dann ist auch L' nicht entscheidbar.

Beweis: Wäre L' entscheidbar, dann wäre auch L entscheidbar.

- Berechne für eine beliebige Eingabe x den Wert $f(x)$
  und entscheide, ob $f(x) \in L'$ gilt.
- Daher wäre mittels f auch entschieden, ob $x \in L$ ist.

Satz: H ist nicht entscheidbar, denn es gilt $\overline{DIAG} \leq H$ mittels der folgenden, berechenbaren Funktion f:

- Falls w keine Gödelisierung einer Turingmaschine ist:
  Bilde w auf $\langle \widetilde{M} \rangle \epsilon$ ab, wobei $\widetilde{M}$ eine feste TM ist, die bei leerer Eingabe hält.

$$w \neq \langle M \rangle \wedge w \in \overline{DIAG} \iff f(w) = \langle \widetilde{M} \rangle \epsilon \in H$$

- Falls $w = \langle M \rangle$ gilt:
  Bilde w auf $\langle M' \rangle \langle M \rangle$ ab, wobei M' die dTM ist, die aus M entsteht, wenn wir jede nicht akzeptierende Rechnung von M zu einer Endlosschleife erweitern.

$$\begin{aligned} w = \langle M \rangle \wedge w \in \overline{DIAG} &\iff M \text{ akzeptiert } \langle M \rangle \\ &\iff M' \text{ gestartet mit } \langle M \rangle \text{ hält} \\ &\iff f(w) = \langle M' \rangle \langle M \rangle \in H \end{aligned}$$

Satz: H_0 ist nicht entscheidbar, denn es gilt $H \leq H_0$:

- Zu $\langle M \rangle x$ sei $\bar{M}(M, x)$ die TM, die gestartet mit leerem Band zuerst x schreibt, und sich dann wie M gestartet mit x verhält.
- f bildet $\langle M \rangle x$ auf $\langle \bar{M}(M, x) \rangle$ ab und ist berechenbar.
- Es gilt: M gestartet mit x hält $\iff$ $\bar{M}(M, x)$ gestartet mit leerem Band hält.

Folgerung: Die Klasse der rekursiv aufzählbaren Sprachen ist nicht gegen Komplementbildung abgeschlossen.

- H ist rekursiv aufzählbar, $\bar{H}$ aber nicht, denn
- wäre $\bar{H}$ rekursiv aufzählbar, dann wäre H entscheidbar.

Weitere nicht entscheidbare Probleme:

- Totalitätsproblem $\textsc{Total} := \{\langle M \rangle \mid M \text{ hält für jeden Input}\}$
- Endlichkeitsproblem $\textsc{Endlich} := \{\langle M \rangle \mid M \text{ hält für endlich viele Inputs}\}$
- Äquivalenzproblem $\textsc{Äquiv} := \{\langle M \rangle, \langle M' \rangle \mid L(M) = L(M')\}$

Allgemein gilt der Satz von Rice: Sei $\mathcal{R}$ die Menge der von dTM berechenbaren partiellen Funktionen und S eine Teilmenge von $\mathcal{R}$ mit $\emptyset \neq S \neq \mathcal{R}$. Dann ist die Sprache

$$L(S) = \{\langle M \rangle \mid M \text{ berechnet eine Funktion aus } S\}$$

nicht entscheidbar.

In anderen Worten: Aussagen über die von einer TM berechneten Funktion sind nicht entscheidbar.

Postsches Korrespondenzproblem:

Gegeben: Eine endliche Menge von Wortpaaren $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_k, y_k)$, wobei $x_i, y_i \in \Sigma^+$ gilt. *Diese Wortpaare sind vorstellbar als verschiedene Dominosteine.*

Gefragt: Gibt es eine Folge von Indizes $i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, k\}$, $n \geq 1$, mit $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$?

Beispiel: $K = ((1, 101), (10, 00), (011, 11))$ besitzt die Lösung (1,3,2,3), denn es gilt

$$\underbrace{1}_{x_1}\underbrace{011}_{x_3}\underbrace{10}_{x_2}\underbrace{011}_{x_3} = \underbrace{101}_{y_1}\underbrace{11}_{y_3}\underbrace{00}_{y_2}\underbrace{11}_{y_3}$$

Ohne Beweis: Das Postsche Korrespondenzproblem ist nicht entscheidbar.

Ablauf bei Reduktionen: Zeige $ALT \leq NEU$ mittels f.

- Erstelle zunächst eine Funktion f, die aus einer Eingabe x von ALT eine Eingabe $f(x)$ von NEU macht, wobei gelten muss:

$$x \in ALT \iff f(x) \in NEU$$

- Wäre NEU berechenbar, dann wäre auch ALT berechenbar.

$$x \overset{?}{\in} ALT \rightsquigarrow \begin{cases} f(x) \in NEU & \Rightarrow \quad x \in ALT \\ f(x) \notin NEU & \Rightarrow \quad x \notin ALT \end{cases}$$

Um $x \overset{?}{\in} ALT$ zu beantworten, nutzen wir also f und den (fiktiven) Algorithmus für NEU.

Übersicht

Komplexitätsklassen

- P vs. NP: Probleme, für die eine polynomiell zeitbeschränkte dTM bzw. nTM existiert.
- L vs. NL: Probleme, für die eine logarithmisch platzbeschränkte dTM bzw. nTM existiert.
- PSPACE vs. NPSPACE: Probleme, für die eine polynomiell platzbeschränkte dTM bzw. nTM existiert.
- Es gibt viele weitere Komplexitätsklassen.

Es gilt folgende Hierarchie:

$$L \subset NL \subset P \subset NP \subset PSPACE = NPSPACE$$

Satz von Savitch: Es gilt $\text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s^2(n))$ für jede platzkonstruierbare Funktion $s : \mathbb{N} \to \mathbb{N}$ mit $s \in \Omega(\log(n))$.

CLIQUE: Finde zu einem gegebenen Graphen $G = (V, E)$ und einem $k \in \mathbb{N}$ eine Teilmenge $V' \subseteq V$ der Knoten, sodass je zwei Knoten in V' durch eine Kante in E verbunden sind und $|V'| \geq k$ gilt.

Algorithmus:

- „Rate“ nichtdeterministisch eine Menge $V' \subseteq V$ mit $|V'| \geq k$
- und verifiziere, ob V' eine Clique ist.

HAMILTON-KREIS: Finde eine Route in einem gegebenen Graphen $G = (V, E)$, die genau einmal durch jeden Knoten in V läuft und wieder am Startknoten endet.

Algorithmus:

- „Rate“ nichtdeterministisch eine Menge $E' \subseteq E$
- und verifiziere, ob E' eine Rundreise ist.

INDEPENDENT SET: Finde in einem gegebenen Graphen $G = (V, E)$ eine Menge $V' \subseteq V$ mit $|V'| \geq k$, sodass für alle $u, v \in V'$ gilt: $\{u, v\} \notin E$.

- „Rate“ nichtdeterministisch eine Menge $V' \subseteq V$ mit $|V'| \geq k$
- und verifiziere, ob V' eine unabhängige Menge ist.

DOMINATING SET: Finde in einem gegebenen Graphen $G = (V, E)$ eine Menge $V' \subseteq V$ mit $|V'| \leq k$, sodass für alle $v \in V$ der Knoten v oder einer seiner Nachbarn in V' enthalten ist.

- „Rate“ nichtdeterministisch eine Menge $V' \subseteq V$ mit $|V'| \leq k$
- und verifiziere, ob V' ein dominierende Menge ist.

VERTEX COVER: Finde in einem gegebenen Graphen $G = (V, E)$ eine Knotenmenge $V' \subseteq V$ mit $|V'| \leq k$, sodass jede Kante $e \in E$ inzident zu einem Knoten in V' ist.

- „Rate“ nichtdeterministisch eine Menge $V' \subseteq V$ mit $|V'| \leq k$
- und verifiziere, ob V' ein Vertex-Cover ist.

Ohne Beweis: Der Nichtdeterminismus kann auf eine initiale „Ratephase“ beschränkt werden.

Übung 11. *Vergleichen Sie Dominating-Set und Vertex-Cover. Welche Beziehung besteht zwischen diesen beiden Begriffen? Ist vielleicht jedes Vertex-Cover auch ein Dominating-Set? Oder ist jedes Dominating-Set auch ein Vertex-Cover?*

Die obigen Probleme können deterministisch sehr einfach, aber (bisher) nicht effizient gelöst werden:

- Zähle alle $\binom{n}{k}$ Teilmengen V' mit $|V'| = k$ auf und teste die gesuchte Eigenschaft. Zur Erinnerung:

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!} \\ &\leq \frac{n \cdot n \cdot \ldots \cdot n}{\cancel{k!}} \leq n^k\end{aligned}$$

- Laufzeit $\mathcal{O}(n^k \cdot (n+m))$ ist für konstantes k polynomiell.
  Dabei ist n die Anzahl der Knoten und m die Anzahl der Kanten.

Frage: Warum wird zusätzlich zu den Graphen auch die Größe k in der Eingabe angegeben?

Wir unterscheiden bei vielen Problemen die Entscheidungs-, Optimierungs- und Suchvariante, hier am Beispiel CLIQUE.

- Entscheidungsvariante: Gibt es in einem gegebenen Graphen $G = (V, E)$ eine Clique der Größe mindestens k? Dabei ist k ein Teil der Eingabe.
- Optimierungsversion: Aus wie vielen Knoten besteht eine Clique maximaler Größe im gegebenen Graphen $G = (V, E)$?
- *Suchvariante:* Finde eine Knotenmenge $V' \subseteq V$ des Graphen $G = (V, E)$, sodass V' eine Clique maximaler Größe ist.

Problem bei der Optimierungs- und der Suchvariante: Es reicht nicht, die „geratene“ Lösung zu verifizieren. Es muss zusätzlich noch sichergestellt sein, dass es keine größere Clique im Graphen gibt. Bei der Entscheidungsvariante ist das nicht nötig.

Siehe auch `http://www.csc.kth.se/~viggo/wwwcompendium/` für weitere schwere Optimierungsprobleme.

Frage: Wie schwer sind die verschiedenen Varianten in Bezug zueinander?

- Wenn wir einen Algorithmus für Dec-Clique kennen, können wir damit eine Lösung für Max-Clique berechnen?
- Können wir mit einem Algorithmus für Max-Clique auch eine Lösung für Search-Clique berechnen?

Dec-Clique bezeichnet die Entscheidungs-, Max-Clique die Optimierungs- und Search-Clique die Suchvariante.

Die anderen Richtungen sind klar:

- Ein Algorithmus für Max-Clique löst auch Dec-Clique: Falls k kleiner oder gleich der optimalen Lösung ist, dann gib ja aus, sonst nein.
- Ein Algorithmus für Search-Clique löst auch Max-Clique: Gib die Größe der berechneten Teilmenge aus.

k-Coloring: Sei $f : V \to \{1, 2, \ldots, k\}$ eine Abbildung der Knoten auf die ersten k natürlichen Zahlen. Dann nennt man f eine *k-Färbung* (k-coloring) eines Graphen $G = (V, E)$, wenn $f(u) \neq f(v)$ für alle Kanten $\{u, v\} \in E$ gilt.

- „Rate“ nichtdeterministisch für den Graphen $G = (V, E)$ eine Zuordnung $f : V \to [k]$ von Farben zu Knoten, wobei $[k] := \{1, 2, \ldots, k\}$ ist.
- Prüfe für jede Kante $\{u, v\} \in E$, ob $f(u) \neq f(v)$ gilt.

Bei diesem Problem muss die Zahl k nicht Teil der Eingabe sein.

- Die obigen Probleme Clique, Independent Set und Dominating Set konnten für konstantes k in polynomieller Zeit $\mathcal{O}(n^k)$ gelöst werden.
- Für das Problem k-Coloring ist bereits für $k = 3$ kein Algorithmus mit polynomieller Laufzeit bekannt, jedenfalls nicht für allgemeine Graphen.

Varianten des Färbungsproblems:

- Entscheidungsvariante: Gibt es für einen gegebenen Graphen $G = (V, E)$ eine Färbung der Knoten mit höchstens k Farben? Dabei ist k Teil der Eingabe.
- Optimierungsversion: Wie viele Farben reichen aus, um den Graphen zu färben?
- *Suchvariante:* Bestimme eine Abbildung $f : V \to [k]$, sodass f eine k-Färbung von G ist und keine Färbung mit weniger Farben existiert.

Drei Varianten können einfach ineinander umgerechnet werden.

- Dec-Col $\to$ Min-Col: Nutze binären Entscheidungsbaum.
- Min-Col $\to$ Dec-Col: Falls k größer oder gleich der optimalen Lösung ist, dann gib ja aus, sonst nein.
- Search-Col $\to$ Min-Col: Gib die Anzahl k der genutzten Farben aus.

Die Umrechnung von Min-Col $\to$ Search-Col ist deutlich komplizierter.

Min-Col → Search-Col

- Sei $G = (V, E)$ ein Graph mit den Knoten $V = \{v_1, \ldots, v_n\}$.
- Berechne zunächst das minimale k, für das eine k-Färbung von G existiert.
- Zum Bestimmen einer k-Färbung nutzen wir eine k-Clique C, die die Knoten $x_1, \ldots, x_k$ enthält.
- C kann mit k vielen Farben gefärbt werden, wobei o.B.d.A. für die Färbung f angenommen werden kann, dass $f(x_i) = i$ gilt.
- Der initiale Graph G_0 besteht aus der Vereinigung von G und C.
- Berechne G_i aus G_{i-1} wie folgt:
  - Für $j := 1 \ldots k$ führe die folgenden zwei Schritte aus:
  - $E(G_i) := E(G_{i-1}) \cup \{(v_i, x) \mid x \in \{x_1, \ldots, x_k\} - \{x_j\}\}$
  - Teste mittels Min-Col, ob G_i immer noch k-färbbar ist. Falls ja, dann ist G_i gefunden, sonst fahre mit dem nächsten Wert von j fort.

Die Färbung können wir aus G_n ablesen: Knoten v_i erhält die Farbe j, wenn v_i nicht mit dem Knoten x_j durch eine Kante verbunden ist.

Für das Problem des Handlungsreisenden kann man auch eine Entscheidungs-, Optimierungs- und Suchvariante definieren.

Auch hier ist es recht einfach möglich, mittels eines Algorithmus für die eine Variante die Lösung einer anderen Variante zu berechnen.

- DEC-TSP $\rightarrow$ MIN-TSP: Nutze binären Entscheidungsbaum.
- MIN-TSP $\rightarrow$ DEC-TSP: Falls k größer oder gleich der optimalen Lösung ist, dann gib ja aus, sonst nein.
- SEARCH-TSP $\rightarrow$ MIN-TSP: Addiere die Werte aller Kanten, die auf der optimalen Tour liegen und gib diese Summe aus.
- MIN-TSP $\rightarrow$ SEARCH-TSP: Erhöhe den Wert einer Kante e um einen Wert c. Falls die optimale Länge unverändert ist, gehört die Kante e nicht zur optimalen Tour. Falls doch, setze den Wert der Kante zurück. Teste auf diese Weise alle Kanten.

Definition: Eine Sprache L ist *polynomiell reduzierbar* auf eine Sprache L', wenn eine Funktion f existiert, mit

- $x \in L \iff f(x) \in L'$
- und f ist in polynomieller Zeit berechenbar.

Analog zur Berechenbarkeit schreiben wir dann $L \leq_p L'$ und sagen: L ist nicht schwieriger zu berechnen als L'.

Beachte: Wir haben Probleme immer als Sprachen definiert, die von Turingmaschinen akzeptiert werden.

$$\begin{aligned}
\textsc{Clique} &:= \{(G,k) \mid \text{Graph } G \text{ enthält Clique } C \text{ mit } |C| \geq k\} \\
\text{HCP} &:= \{G \mid \text{Graph } G \text{ enthält einen Hamilton-Kreis}\} \\
\text{ISP} &:= \{(G,k) \mid G \text{ enthält Independent-Set } I \text{ mit } |I| \geq k\} \\
\text{DSP} &:= \{(G,k) \mid G \text{ enthält Dominating-Set } D \text{ mit } |D| \leq k\} \\
\text{VCP} &:= \{(G,k) \mid G \text{ enthält Vertex-Cover } C \text{ mit } |C| \leq k\}
\end{aligned}$$

Eine solche Definition würde ohne Angabe von k gar keinen Sinn ergeben.

Hinweis: Polynomielle Reduktion ist transitiv.

$$L_1 \leq_p L_2 \text{ und } L_2 \leq_p L_3 \;\Rightarrow\; L_1 \leq_p L_3$$

Idee: Sei $L_1 \leq_p L_2$ mittels f in Zeit p, und sei $L_2 \leq_p L_3$ mittels g in Zeit q. Dann gilt $L_1 \leq_p L_3$ mittels $g \circ f$:

$$x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3$$

$g(f(x))$ ist in polynomieller Zeit berechenbar:

- $f(x)$ hat polynomielle Länge $p(|x|)$, da eine Turingmaschine in Zeit $p(|x|)$ nur $p(|x|)$ Zeichen schreiben kann.
- Also kann $g(f(x))$ in Zeit $p(|x|) + q(p(|x|))$ berechnet werden, also in polynomieller Zeit.

Definition: Ein Problem L ist NP-vollständig, wenn gilt:

- Es existiert eine nTM, die das Problem in polynomieller Zeit löst, also $L \in \text{NP}$.
- Alle $L' \in \text{NP}$ sind polynomiell reduzierbar auf L, also $L' \leq_p L$.

$\rightarrow$ Also gehört L zu den schwierigsten Problemen in NP.

Satz: Ist auch nur ein einziges NP-vollständiges Problem deterministisch in polynomieller Zeit lösbar, dann sind alle NP-vollständigen Probleme auch deterministisch in polynomieller Zeit lösbar!

Übung 12. *Beweisen Sie den letzten Satz.*

Leider ist bis heute für kein einziges NP-vollständiges Problem ein polynomiell zeitbeschränkter (deterministischer) Algorithmus bekannt.

SAT ist NP-vollständig

Frage: Wie findet man ein erstes NP-vollständiges Problem?

Erfüllbarkeitsproblem (SATisfiability):

- Gegeben: Eine Formel F der Aussagenlogik.
- Gefragt: Ist die Formel F erfüllbar?
- $\text{SAT} := \{F \mid F \text{ ist erfüllbare aussagenlogische Formel}\}$

Satz von Cook[3]*:* SAT ist NP-vollständig

Wir werden im folgenden zeigen:

- $\text{SAT} \in \text{NP}$
- $\forall L \in NP: \; L \leq_p \text{SAT}$

[3] aus dem Buch von Uwe Schöning: Theoretische Informatik – kurz gefasst.

SAT $\in NP$

- Maschine M läuft einmal über die Eingabe und stellt fest, welche Variablen in der Formel F vorkommen – dies seien $x_1, \ldots, x_k$.
- In der eigentlichen nichtdeterministischen Phase „rät“ die Maschine M eine Belegung $a_1, \ldots, a_k$ mit $a_i \in \{0, 1\}$.

  Zu diesem Zeitpunkt existieren 2^k mögliche unabhängige Rechnungen.
- M berechnet den Wert von F unter der betreffenden Belegung und akzeptiert genau dann, wenn der Wert 1 ist.
- $F \in$ SAT genau dann, wenn es eine Rechnung von M gibt, die F akzeptiert.
- Da $k \leq |F|$ ist, ist die Rechenzeit polynomial.

$\forall L \in NP: \; L \leq_p \text{SAT}$:

- Sei M eine nTM, die L in polynomieller Zeit entscheidet.
- Sei $\Gamma = \{a_1, \ldots, a_\ell\}$ das Arbeitsalphabet von M.
- Sei $Z = \{z_1, \ldots, z_k\}$ die Zustandsmenge von M.
- Die δ-Relation von M enthalte die Zeile $\delta(z_e, a) \ni (z_e, a, N)$.
  $\rightarrow$ Ein einmal erreichter Endzustand wird also nie mehr verlassen.
- Sei p ein Polynom, das die Laufzeit von M beschreibt; $p(n)$ ist also eine obere Schranke für die Laufzeit von M bei einer Eingabe der Größe n.
- Sei $x = x_1 x_2 \ldots x_n \in \Sigma^*$ eine Eingabe für M.

Wir werden nun eine Boolesche Formel F in Abhängigkeit von x angeben, sodass gilt:

$$x \in L \iff F(x) \text{ ist erfüllbar}$$

Die gesuchte Formel F enthält folgende Boolesche Variablen:

Variable	Indizes	Bedeutung
$zust_{t,z}$	$t = 0, \ldots, p(n)$ und $z \in Z$	$zust_{t,z} = 1 \iff$ nach t Schritten befindet sich M im Zustand z
$pos_{t,i}$	$t = 0, \ldots, p(n)$ und $i = -p(n), \ldots, p(n)$	$pos_{t,i} = 1 \iff$ der Schreib-Lesekopf von M befindet sich nach t Schritten auf Position i
$band_{t,i,a}$	$t = 0, \ldots, p(n)$ und $i = -p(n), \ldots, p(n)$ und $a \in \Gamma$	$band_{t,i,a} = 1 \iff$ nach t Schritten befindet sich auf Bandposition i das Zeichen a

Dabei ist n die Länge der Eingabe und p das Polynom, das die Laufzeit von M beschreibt.

Die Formel F ist aus mehreren Teilformeln aufgebaut:

$$F = R \wedge A \wedge U_1 \wedge U_2 \wedge E$$

- R: Randbedingungen
- A: Anfangsbedingungen
- U_1 und U_2: Übergangsbedingungen
- E: Endbedingungen

Im Folgenden wird immer wieder die Teilformel $G(y_1, \ldots, y_k)$ vorkommen:

$$G(y_1, \ldots, y_k) = \left(\bigvee_{i=1}^{k} y_i \right) \wedge \left(\bigwedge_{i \neq j} (\neg y_i \vee \neg y_j) \right)$$

wird genau dann wahr, wenn genau eins der y_i wahr ist.

Randbedingung R drückt aus:

- M ist zu jedem Zeitpunkt t in genau einem Zustand:

$$\bigwedge_t G(zust_{t,z_1}, \ldots, zust_{t,z_k})$$

- Der Kopf von M befindet sich zu jedem Zeitpunkt t an genau einer Bandposition:

$$\bigwedge_t G(pos_{t,-p(n)}, \ldots, pos_{t,p(n)})$$

- Zu jedem Zeitpunkt t und an jeder Bandposition i steht genau ein Zeichen:

$$\bigwedge_{t,i} G(band_{t,i,a_1}, \ldots, band_{t,i,a_\ell})$$

Anfangsbedingung A drückt für $t = 0$ aus:

- M ist im Anfangszustand z_0.
- Der Schreib-Lesekopf befindet sich auf dem ersten Zeichen.
- Die ersten n Bandpositionen enthalten die Eingabe $x_1, \ldots, x_n$.
- An allen anderen Bandpositionen steht ein Blank.

$$A = zust_{0,z_0} \quad \wedge \quad pos_{0,1} \quad \wedge \bigwedge_{j=1}^{n} band_{0,j,x_j}$$

$$\wedge \bigwedge_{j=-p(n)}^{0} band_{0,j,\square} \quad \wedge \bigwedge_{j=n+1}^{p(n)} band_{0,j,\square}$$

U_1 beschreibt den Übergang vom Zeitpunkt t nach $t+1$ an der Stelle, wo sich der Schreib-Lesekopf befindet:

$$U_1 = \bigwedge_{t,z,i,a} \Big[(zust_{t,z} \wedge pos_{t,i} \wedge band_{t,i,a}) \rightarrow \bigvee_{\delta(z,a) \ni (z',a',y)} (zust_{t+1,z'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'}) \Big]$$

wobei $y \in \{-1, 0, +1\}$ angenommen wird.

U_2 drückt aus, dass sich der Bandinhalt auf allen anderen Positionen nicht ändert:

$$U_2 = \bigwedge_{t,i,a} \Big((\neg pos_{t,i} \wedge band_{t,i,a}) \rightarrow band_{t+1,i,a} \Big)$$

E prüft, ob ein Endzustand erreicht wird. Nach unserer Annahme wird dies insbesondere zum Zeitpunkt $p(n)$ erreicht.

$$E = \bigvee_{z \in E} zust_{p(n),z}$$

Übung 13.

- *Zeigen Sie:* $x \in L \iff F(x)$ *ist erfüllbar*
- *Zeigen Sie, dass alle Teilfolgen polynomielle Länge haben und daher die Reduktion in polynomieller Zeit berechenbar ist.*

NP-vollständige Probleme

Uns interessiert: Was macht ein Problem schwierig?

- Wir beschränken uns auf Formeln in konjunktiver Normalform:
  Ist KNF-SAT auch NP-vollständig?
- Wir beschränken uns auf Formeln in disjunktiver Normalform:
  Ist DNF-SAT auch NP-vollständig?
- Wir beschränken die Anzahl der Variablen pro Klausel auf 3:
  Ist 3KNF-SAT auch NP-vollständig?
- Wir beschränken die Anzahl der Variablen pro Klausel weiter:
  Ist 2KNF-SAT auch NP-vollständig?
- Wir erlauben in jeder Klausel nur eine positive Variable:
  Ist HORN-SAT auch NP-vollständig?

Ist KNF-SAT auch NP-vollständig?

- Jede Boolesche Formel ist äquivalent in konjunktive Normalform umformbar.
- Aber: Das Verfahren hat exponentiellen Aufwand.

Wir zeigen: $\text{SAT} \leq_p \text{3KNF-SAT}$

- Wir geben ein polynomielles Verfahren an, das beliebige Boolesche Formeln F umwandelt in F'. Dabei ist F' in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel und es gilt:

$$F \text{ ist erfüllbar} \iff F' \text{ ist erfüllbar}$$

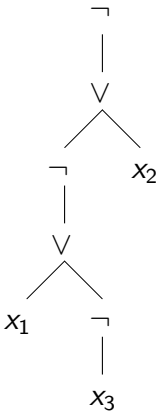
- Es ist nur Erfüllbarkeitsäquivalenz zwischen F und F' verlangt, nicht Äquivalenz im strengen Sinne.

Wir stellen uns eine Formel als Baumstruktur vor.

Beispiel: $F = \neg(\neg(x_1 \vee \neg x_3) \vee x_2)$

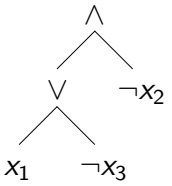
1. Schritt: Wir wenden deMorgan's Regeln an und bringen alle Negationszeichen zu den Blättern.

Dabei ändert sich ggf. ein $\vee$-Knoten zu einem $\wedge$-Knoten und umgekehrt. Dieser Schritt erfordert nur einen Durchlauf über die Formel.



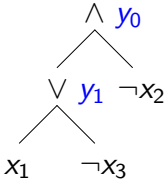
Nach dem ersten Schritt kommen im Innern des Baumes nur Knoten vom Typ $\vee$ und $\wedge$ vor und die Blätter sind nur mit negierten oder nicht-negierten Variablen beschriftet.

Diese so entstandene Formel ist äquivalent zu der ursprünglichen Formel.

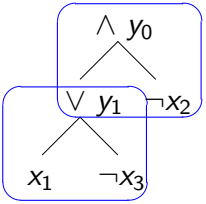


2. Schritt: Wir ordnen jedem inneren Knoten eine neue Variable aus $\{y_0, y_1, \ldots\}$ zu.

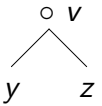
Der Baumwurzel wird y_0 zugewiesen.



3. Schritt: Wir fassen jede Verzweigung zu einer Dreier-Gruppe zusammen.



Jede Verzweigung der Bauart



mit $\circ \in \{\wedge, \vee\}$ ordnen wir eine Teilformel der Form

$$(v \leftrightarrow (y \circ z))$$

zu. Alle diese Formeln werden mit $\wedge$ verknüpft, zusätzlich kommt die Teilformel y_0 hinzu. Dadurch erhalten wir die Formel F_1:

$$F_1 = (y_0) \wedge (y_0 \leftrightarrow (y_1 \wedge \neg x_2)) \wedge (y_1 \leftrightarrow (x_1 \vee \neg x_3))$$

Diese so entstandene Formel ist nur noch erfüllbarkeitsäquivalent zu der ursprünglichen Formel.

4. Schritt: Jede der Teilformeln wird separat in konjunktive Normalform umgeformt:

- In jeder Teilformel kommen nur 3 Literale vor.
- Der exponentielle Aufwand für das Umformen in konjunktive Normalform spielt keine Rolle mehr, da die Teilformeln nur konstante Größe haben.

Umwandeln der Teilformeln:

$$\begin{aligned} (a \leftrightarrow (b \vee c)) &\mapsto (a \vee \neg b) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg c) \\ (a \leftrightarrow (b \wedge c)) &\mapsto (\neg a \vee b) \wedge (\neg a \vee c) \wedge (a \vee \neg b \vee \neg c) \end{aligned}$$

Alle Umformungsschritte können mit polynomiellem Aufwand erfolgen.

Daher ist $\text{SAT} \leq_p 3\text{KNF-SAT}$ gezeigt!

Übung 14. *Zeigen Sie die Äquivalenz der obigen Aussagen.*

3Knf-Sat ist NP-vollständig, also auch Knf-Sat, da die Einschränkung auf 3 Literale pro Klausel ja gerade ein Spezialfall von Knf-Sat ist.

Satz: 2Knf-Sat ist in P.

Beweis: Sei F eine 2Knf-Formel über den Variablen $x_1, \ldots, x_n$. Dann definieren wir einen gerichteten Graphen $G = (V, E)$ mit

- $V = \{x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n\}$ und
- $E = \{(\neg x_i, x_j) \mid (x_i \vee x_j) \in F\} \cup \{(\neg x_j, x_i) \mid (x_i \vee x_j) \in F\}$

Übung 15. *Zeigen Sie: F ist genau dann nicht erfüllbar, wenn der Graph einen Kreis enthält, auf dem sowohl x_i als auch $\neg x_i$ liegt.*

Hinweis: Die Klausel $(x_i \vee x_j)$ kann nur dann erfüllt sein, wenn x_i oder x_j mit 1 belegt sind, oder anders ausgedrückt: $\neg x_i \Rightarrow x_j$ oder $\neg x_j \Rightarrow x_i$ muss gelten, beides ist äquivalent zu $x_i \vee x_j$.

Eine Formel F in konjunktiver Normalform ist eine Hornformel, wenn jede Klausel in F höchstens ein positives Literal enthält.

Beispiel:

$$F = (a \vee \neg b) \wedge (\neg c \vee \neg a \vee d) \wedge (\neg a \vee \neg b) \wedge d \wedge \neg e$$

Hornformeln können anschaulich als Konjunktionen von Implikationen geschrieben werden.

$$F \equiv (b \to a) \wedge (c \wedge a \to d) \wedge (a \wedge b \to 0) \wedge (1 \to d) \wedge (e \to 0)$$

Machen Sie sich klar, dass die obigen Formeln äquivalent sind.

Satz: HORN-SAT ist in P.

Beweis: Sei F eine Hornformel über den atomaren Formeln $x_1, \ldots, x_n$. Der folgende Algorithmus markiert die Variablen, die mit 1 belegt werden müssen.

1. Für alle Teilformeln $1 \to x_i$ aus F markiere Literal x_i.
2. Wiederhole:
   - Markiere y, falls es eine Teilformel $(x_{i_1} \wedge \ldots \wedge x_{i_k}) \to y$ gibt, bei der $x_{i_1}, \ldots, x_{i_k}$ bereits markiert sind.
   - Falls es eine Teilformel $(x_{i_1} \wedge \ldots \wedge x_{i_k}) \to 0$ gibt, bei der $x_{i_1}, \ldots, x_{i_k}$ bereits markiert sind, gib *unerfüllbar* aus und stoppe.
3. Gib *erfüllbar* aus und stoppe.

Alle nicht-markierten Variablen können mit 0 belegt werden.

Übung 16.

- *Wenden Sie den Algorithmus auf die Formel aus dem obigen Beispiel an.*
- *Zeigen Sie, dass der Algorithmus nach spätestens n Markierungsschritten hält.*
- *Zeigen Sie, dass der Algorithmus korrekt ist.*

Satz: DNF-SAT ist in P.

Beweis: Sei $F = (x_{1,1} \wedge \ldots \wedge x_{1,k_1}) \vee \ldots \vee (x_{m,1} \wedge \ldots \wedge x_{m,k_m})$ eine Formel in disjunktiver Normalform. Dann gilt:

- F ist genau dann erfüllbar, wenn wenigstens eins der m Konjunktionsglieder $(x_{j,1} \wedge \ldots \wedge x_{j,k_j})$ erfüllbar ist.
- Ein Konjunktionsglied $(x_{j,1} \wedge \ldots \wedge x_{j,k_j})$ ist genau dann unerfüllbar, wenn es eine Variable x und $\neg x$ enthält.

Satz: CLIQUE ist NP-vollständig.

Beweis: $3\text{KNF-SAT} \leq_p \text{CLIQUE}$

- Sei F eine Formel in konjunktiver Normalform mit m Klauseln zu je genau 3 Literalen. (Wir können der Einfachheit halber genau 3 Literale annehmen, indem wir Literale in einer Klausel verdoppeln.)
- Wir konstruieren den Graphen G mit $3m$ Knoten und:

$$G \text{ hat Clique der Größe } m \iff F \text{ ist erfüllbar}$$

- Sei $F = (a_{11} \vee a_{12} \vee a_{13}) \wedge \cdots \wedge (a_{m1} \vee a_{m2} \vee a_{m3})$
- Sei $G = (V, E)$ mit
  - $V = \{(i,j) \mid 1 \leq i \leq m,\ 1 \leq j \leq 3\}$ und
  - $E = \{\{(i,j),(k,l)\} \mid i \neq k \text{ und } a_{ij} \neq \neg a_{kl}\}$.

Also:

- Die Knoten von G sind die in F vorkommenden Literale.
- Es gibt jeweils einen Knoten für jedes Vorkommen eines Literals in einer Klausel.
- Zwei Knoten sind miteinander verbunden, wenn ihre Literale
  - in verschiedenen Klauseln sind und
  - sich nicht direkt widersprechen (x und $\neg x$).
- Die Größe von G ist polynomiell beschränkt: Es gibt $3m$ Knoten, und damit höchstens $9m^2$ viele Kanten.

Übung 17. *Zeigen Sie, dass F genau dann erfüllbar ist, wenn G eine Clique der Größe m hat.*

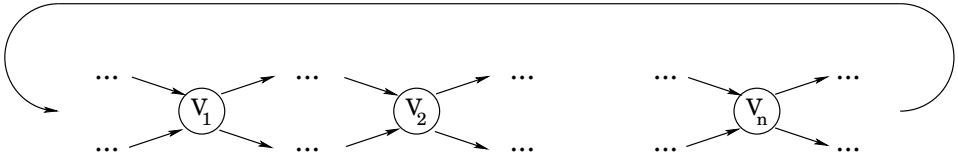
Satz: Gerichteter Hamilton-Kreis ist NP-vollständig.

Beweis: 3Knf-Sat $\leq_p$ Gerichteter Hamilton-Kreis

- Sei F eine Formel in konjunktiver Normalform mit m Klauseln zu je genau 3 Literalen. Also:

$$F = (a_{11} \vee a_{12} \vee a_{13}) \wedge \cdots \wedge (a_{m1} \vee a_{m2} \vee a_{m3})$$

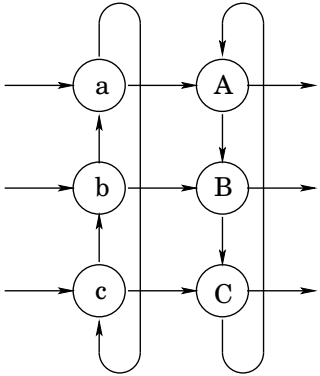
- Seien $x_1, x_2, \ldots, x_n$ die Variablen von F.
- Der Graph G hat zunächst einmal die Knoten $v_1, \ldots, v_n$, die die Variablen von F repräsentieren.



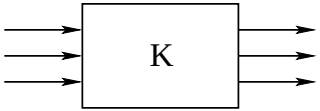
Von jedem Knoten v_i gehen zwei Kanten aus. Diese führen zu Klauselgraphen K, von denen es m Kopien $K_1, \ldots, K_m$ gibt.

Anmerkungen:

- Klauseln, die sowohl eine Variable x_i als auch die negierte Variable $\neg x_i$ enthalten, sind erfüllt und können aus der Formel gestrichen werden.
- Kommt eine Variable x_i entweder nur positiv oder nur negativ in der Formel vor, dann können alle Klauseln, die x_i enthalten, aus der Formel gestrichen werden.



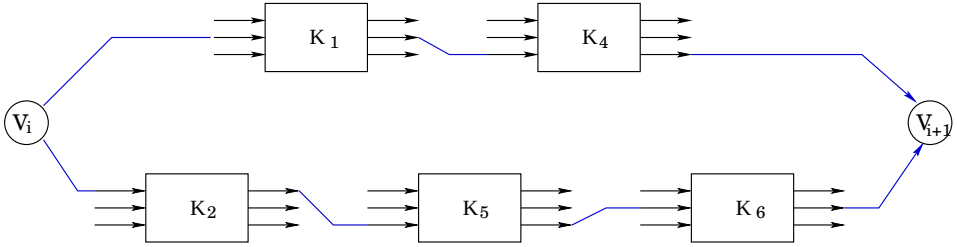
Im Folgenden stellen wir diese Klauselgraphen durch nebenstehendes Symbol dar:



Der obere vom Knoten v_i ausgehende Weg orientiert sich an den Vorkommen von x_i in den Klauseln, der untere an denen von $\neg x_i$.

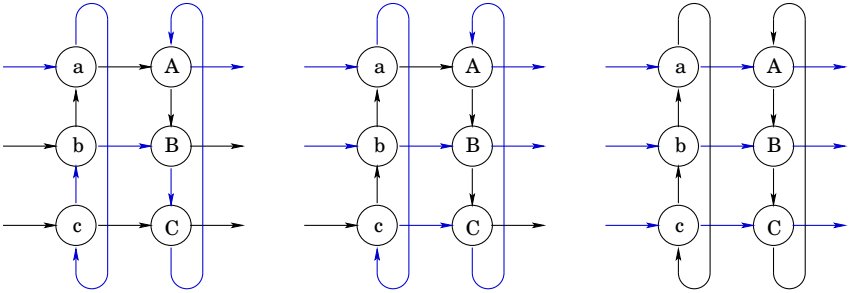
Beispiel:

- x_i komme in Klausel 1, Position 2 und in Klausel 4, Position 3 vor.
- $\neg x_i$ komme in Klausel 2, Position 1, in Klausel 5, Position 3 und in Klausel 6, Position 2 vor.
- Dann erhalten wir folgende Kanten im Graphen:



Wenn die Formel F eine erfüllende Belegung hat:

- Wenn x_i die Belegung 1 hat, so folge von v_i aus dem oberen Pfad, sonst dem unteren.
- Dann durchläuft der Pfad die entsprechenden Klauselgraphen K_j, in denen x_i bzw. $\neg x_i$ vorkommt.
- Je nachdem, wie viele und welche Literale in Klausel j den Wert 1 haben, wird der Klauselgraph K_j in einer von sieben Arten durchlaufen. Daraus ergibt sich der Hamilton-Kreis.



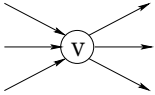
Wenn der Graph einen Hamilton-Kreis besitzt:

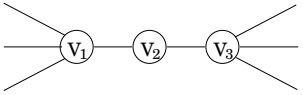
- Wenn der Kreis bei Knoten a einen Klauselgraphen betritt, dann wird dieser Klauselgraph bei Knoten A verlassen. (gilt ebenso für Knotenpaar b,B und c,C)
- Ein bei a eintretender Hamilton-Kreis kann nur folgende Wege nehmen:
  - a $\rightarrow$ A oder
  - a $\rightarrow$ c $\rightarrow$ C $\rightarrow$ A oder
  - a $\rightarrow$ c $\rightarrow$ b $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A
- Der Hamilton-Kreis kann also nur wie vorgesehen durch die Klauselgraphen laufen.
- Eine erfüllende Belegung ergibt sich anhand dessen, ob ein Knoten v_i oben oder unten verlassen wird.

Übung 18. *Wie sieht eine nicht-erfüllbare Formel in KNF aus?*

Satz: GERICHTETER HAMILTON-KREIS $\leq_p$ HAMILTON-KREIS

Beweis:

- Ersetze 

durch 

.
- Dann gilt: Der gerichtete Graph hat genau dann einen gerichteten Hamilton-Kreis, wenn der ungerichtete Graph einen Hamilton-Kreis hat.
- Die Reduktionsfunktion ist in polynomieller Zeit berechenbar.

Euler-Kreis: Eine Rundreise durch den Graphen, bei der jede Kante genau einmal benutzt wird.

Übung 19. *Ist das Euler-Kreis-Problem NP-vollständig?*

Übung 20. *Zeigen Sie, dass die folgenden Probleme NP-vollständig sind:*

- *Hamilton-Pfad: Gegeben ein Graph $G = (V, E)$.*
  Gibt es einen einfachen Weg in G, der jeden Knoten aus V genau einmal besucht?
- *Längster einfacher Weg: Gegeben ein gewichteter Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \to \mathbb{R}$ und ein $k \in \mathbb{R}$.*
  Gibt es einen einfachen Weg in G, der mindestens die Länge k hat?
  Die Länge eines Wegs $p = (v_0, v_1, \ldots, v_n)$ ist dabei definiert als Summe der Kantengewichte $\sum_{i=1}^{n} c(\{v_{i-1}, v_i\})$.
- *Kürzester einfacher Weg: Gegeben ein gewichteter Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \to \mathbb{R}$ und ein $k \in \mathbb{R}$.*
  Gibt es einen einfachen Weg in G, der höchstens die Länge k hat?

Im Bachelorstudium haben wir Kürzeste-Wege-Algorithmen von Dijkstra oder Bellman-Ford kennengelernt, die eine polynomielle Laufzeit haben. Ist das nicht ein Widerspruch zur vorherigen Aussage?

3-Knotenfärbung ist NP-vollständig

3-Knotenfärbung ist in NP:

- Bestimme für Graph $G = (V, E)$ nicht-deterministisch eine Zuordnung $f : V \to \{1, 2, 3\}$ von Farben zu Knoten.
- Prüfe für jede Kante $\{u, v\} \in E$, ob $f(u) \neq f(v)$ gilt.

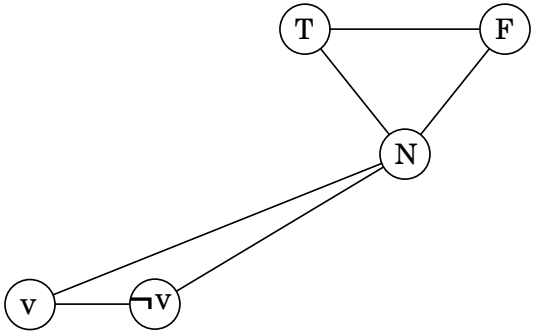
$3\text{KNF-SAT} \leq_p$ 3-Knotenfärbung

- Gegeben: Boolesche Formel $F = C_1 \wedge \ldots \wedge C_m$ in KNF über den Variablen $x_1, \ldots, x_n$ und jede Klausel C_i enthalte nur drei Literale.
- Aufgabe: Erstelle Graph G_F, der genau dann 3-färbbar ist, wenn F erfüllbar ist.
- Idee: Eine Belegung der Variablen muss durch eine 3-Färbung des Graphen erzeugt werden.

3-Knotenfärbung ist NP-vollständig

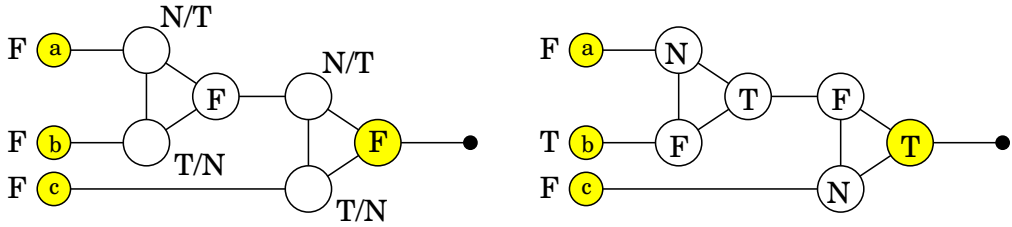
Erzeuge einen Kreis der Länge drei mit Knoten T,F und N.

Erzeuge für jede Variable x_i zwei Knoten v_i und $\neg v_i$. Ordne die Knoten v_i, $\neg v_i$ und N in einem Kreis der Länge drei an.



Wird v_i mit T gefärbt, dann muss $\neg v_i$ mit F gefärbt werden und umgekehrt. Eine 3-Färbung entspricht also einer Belegung der Variablen.

Erzeuge für jede Klausel $C_i = (a \vee b \vee c)$ einen OR-Gadget-Graph:



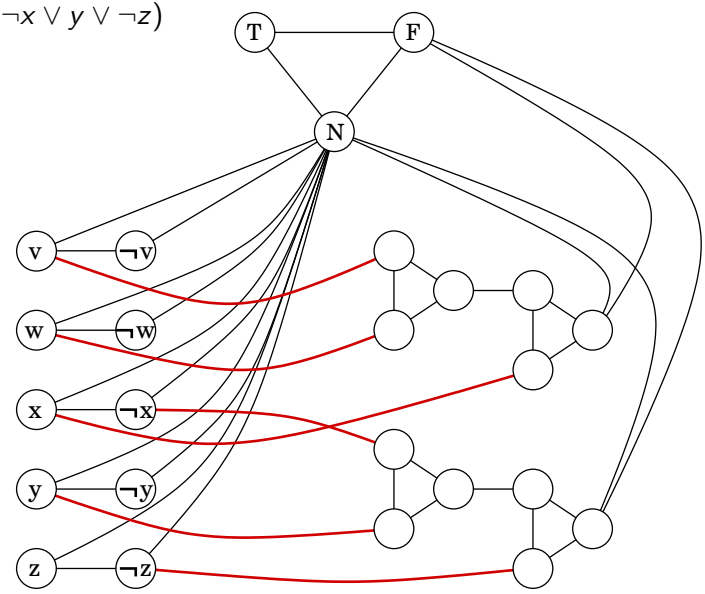
Eigenschaften:

- Falls a, b, c alle mit F gefärbt sind, muss der Ausgang auch mit F gefärbt werden.
- Falls ein Knoten aus a, b, c mit T gefärbt ist, kann der Ausgang auch mit T gefärbt werden.

Der Ausgang eines OR-Gadget-Graphen wird mit den zentralen Knoten N und F jeweils durch eine Kante verbunden.

3-Knotenfärbung ist NP-vollständig

Beispiel: $F = (v \vee w \vee x) \wedge (\neg x \vee y \vee \neg z)$



3-Knotenfärbung ist NP-vollständig

Falls F erfüllbar ist, dann ist G_F mit drei Farben färbbar:

- Falls x_i mit true belegt ist, dann färbe v_i mit T und $\neg v_i$ mit F.
- In jeder Klausel $C_i = (a \vee b \vee c)$ ist mindestens ein Literal a, b, c mit T gefärbt.
- Daher kann der entsprechende OR-Gadget-Graph von C_i mit drei Farben gefärbt werden, sodass der Ausgang mit T gefärbt ist.

Falls G_F mit drei Farben färbbar ist, dann ist F erfüllbar:

- Wenn v_i mit T gefärbt ist, dann belege x_i mit true, sonst mit false. So erhalten wir eine zulässige Belegung der Variablen.
- Für jede beliebige Klausel $C_i = (a \vee b \vee c)$ gilt: Nicht alle Literale a, b, c können mit F gefärbt sein, denn dann müsste der Ausgang mit F gefärbt werden, aber der Ausgang ist mit den Knoten N und F verbunden!

4-Knotenfärbung ist NP-vollständig

4-Knotenfärbung ist in NP:

- Bestimme für Graph $G = (V, E)$ nicht-deterministisch eine Zuordnung $f : V \to \{1, 2, 3, 4\}$ von Farben zu Knoten.
- Prüfe für jede Kante $\{u, v\} \in E$, ob $f(u) \neq f(v)$ gilt.

3-Knotenfärbung $\leq_p$ 4-Knotenfärbung

- Gegeben: Graph $G = (V, E)$
- Aufgabe: Erstelle Graph $G' = (V', E')$, der genau dann 4-färbbar ist, wenn G 3-färbbar ist.
- Idee: Füge einen neuen Knoten x hinzu und verbinde x mit jedem Knoten aus V durch eine Kante, also $V' = V \cup \{x\}$, $E' = E \cup \{\{x, v\} \mid v \in V\}$.

Aufgrund dieser Idee gilt allgemein: k-Färbung ist NP-vollständig für $k \geq 3$.

Ablauf bei Reduktionen: Zeige $ALT \leq_p NEU$ mittels f.

- Erstelle zunächst eine Funktion f, die aus einer Eingabe x von ALT eine Eingabe $f(x)$ von NEU macht, wobei gelten muss:

$$x \in ALT \iff f(x) \in NEU$$

- Wäre NEU effizient lösbar, dann wäre auch ALT effizient lösbar:

$$x \stackrel{?}{\in} ALT \rightsquigarrow \begin{cases} f(x) \in NEU & \Rightarrow \quad x \in ALT \\ f(x) \notin NEU & \Rightarrow \quad x \notin ALT \end{cases}$$

Um $x \stackrel{?}{\in} ALT$ zu beantworten, nutzen wir f und den (fiktiven) effizienten Algorithmus für NEU.

Vertex Cover Problem: Finde zu einem Graphen $G = (V, E)$ eine möglichst kleine Knotenmenge $V' \subseteq V$, sodass jede Kante $e \in E$ inzident zu einem Knoten in V' ist.

$\mathrm{VCP} := \{(G, k) \mid G \text{ hat Vertex-Cover } C \text{ mit } |C| \leq k\}$

Zeigen Sie: $\textsc{Clique} \leq_p \mathrm{VCP}$

Traveling Salesperson Problem: Bestimme zu einem Graphen $G = (V, E, c)$ eine Rundreise, die genau einmal durch alle Knoten führt, die wieder am Startknoten endet und deren Länge möglichst kurz ist. Dabei ist $c : E \to \mathbb{N}$ eine Kostenfunktion, die zu jeder Kante in E deren „Länge“ definiert.

$\mathrm{TSP} := \{(G, k) \mid G \text{ hat Rundreise } P \text{ mit } \sum_{e \in P} c(e) \leq k\}$

Zeigen Sie: $\mathrm{HCP} \leq_p \mathrm{TSP}$

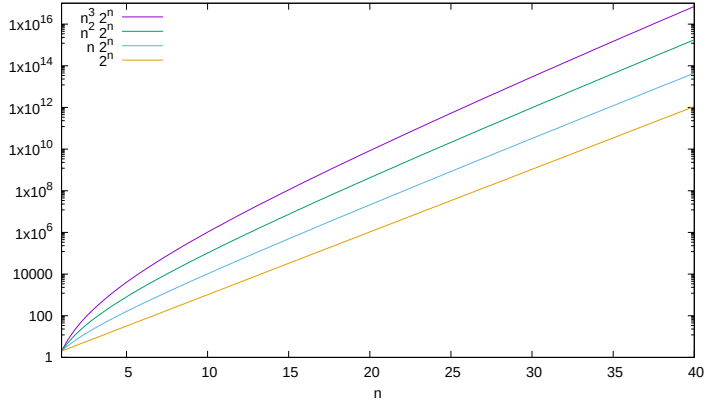
Größe eines Problems

Bestimmend für die Rechenzeit ist

- die *Anzahl* der zu bearbeitenden Elemente
  - Sortieren von Elementen
  - Skalar-Produkt zweier Vektoren
  - Städte-Rundreise
- oder die *Größe* der zu bearbeitenden Elemente
  - Multiplikation zweier Zahlen
  - Primzahlzerlegung einer Zahl
  - Rucksack-Problem
- oder beides.

Übersicht

Bei der $\mathcal{O}^*$-Notation werden auch polynomielle Faktoren vernachlässigt. So gilt $n^2 \cdot 2^n \in \mathcal{O}^*(2^n)$ und auch $n^5 + n^3 \cdot 2^n \in \mathcal{O}^*(2^n)$. Wird manchmal auch als $\tilde{\mathcal{O}}$ bezeichnet.



Idee: Das exponentielle Wachstum ist so stark, dass polynomielle Faktoren kaum ins Gewicht fallen.

Übersicht

Algorithmus A1: Sei x eine Variable in der Formel Φ, und sei $\Phi \mid x$ die Formel, die aus Φ entsteht, wenn x mit 1 belegt wird.

function $3\textsc{Sat}(\Phi)$
  if $3\textsc{Sat}(\Phi \mid x)$ **then return** true
  return $3\textsc{Sat}(\Phi \mid \overline{x})$

Laufzeit: $T(n) = 2 \cdot T(n-1) + poly(n) \in \mathcal{O}^*(2^n)$

Idee zur Laufzeitbestimmung: Wir gehen davon aus, dass sich eine exponentielle Laufzeit $T(n) \in \mathcal{O}^*(b^n)$ ergibt und wollen die Basis b bestimmen. Setze $T(n) := b^n$ und schätze ab:

$$\begin{aligned} b^n &\approx 2 \cdot b^{n-1} + n^k \quad | \quad : b^{n-1} \\ \iff b &\approx 2 + \frac{n^k}{b^{n-1}} \quad | \quad \lim_{n\to\infty} \frac{n^k}{b^{n-1}} = 0 \\ \iff b &\approx 2 \end{aligned}$$

Algorithmus A2: Sei $(x \vee y \vee z)$ eine Klausel in der Formel Φ, und sei $\Phi \mid x$ die Formel, die aus Φ entsteht, wenn x mit 1 belegt wird. [4]

function $3\text{Sat}(\Phi)$
  if $3\text{Sat}(\Phi \mid x)$ **then return** true
  if $3\text{Sat}(\Phi \mid \overline{x}y)$ **then return** true
  return $3\text{Sat}(\Phi \mid \overline{x}\overline{y}z)$

Laufzeit: $T(n) = T(n-1) + T(n-2) + T(n-3) + poly(n) \in \mathcal{O}^*(1,84^n)$

Idee zur Laufzeitbestimmung: Setze $T(n) := b^n$ und schätze ab:

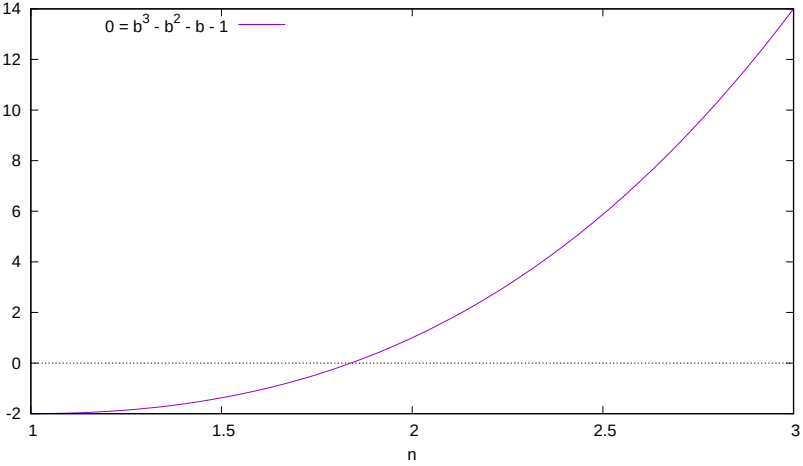
$$\begin{aligned}
& b^n && \approx && b^{n-1} + b^{n-2} + b^{n-3} + n^k && \mid : b^{n-3} \\
\iff & b^3 && \approx && b^2 + b + 1 + \frac{n^k}{b^{n-3}} && \mid \lim_{n\to\infty} \frac{n^k}{b^{n-3}} = 0 \\
\iff & b^3 && \approx && b^2 + b + 1 \\
\iff & 0 && \approx && b^3 - b^2 - b - 1 && \rightsquigarrow \text{Newton-Verfahren}
\end{aligned}$$

[4] Monien, Speckenmeyer: Solving satisfiability in less than 2^n steps: Discrete Applied Mathematics 10 (1985), 287-295.

Das Newton-Verfahren liefert für das charakteristische Polynom

$$b^3 - b^2 - b - 1$$

eine Nullstelle bei $b = 1,8392....$



Algorithmus A3: Wenn eine Variable nur positiv oder nur negativ in der Formel vorkommt, dann können wir den Wert der Variablen entsprechend setzen und die Teilformeln streichen. Seien daher die Klauseln $(x \vee y \vee z)$ und $(\overline{x} \vee u \vee v)$ in der Formel Φ enthalten.

function $3\text{Sat}(\Phi)$
  if $3\text{Sat}(\Phi \mid xu)$ **then return** true
  if $3\text{Sat}(\Phi \mid x\overline{u}v)$ **then return** true
  if $3\text{Sat}(\Phi \mid \overline{x}y)$ **then return** true
  return $3\text{Sat}(\Phi \mid \overline{x}\overline{y}z)$

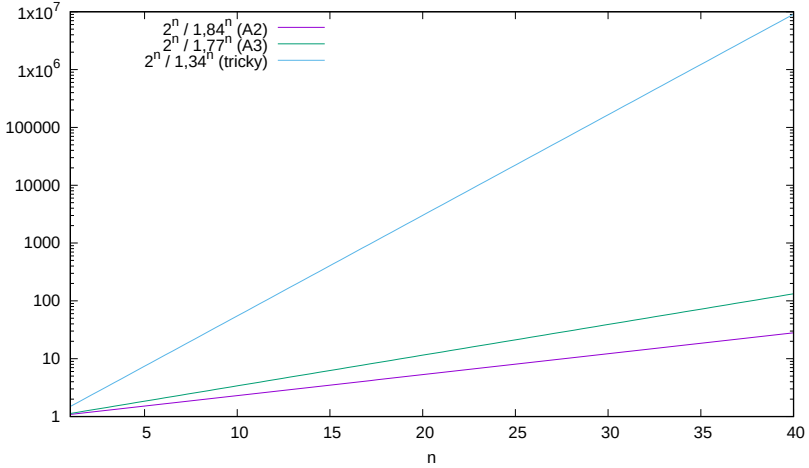
Laufzeit: $T(n) = 2T(n-2) + 2T(n-3) + poly(n) \in \mathcal{O}^*(1, 77^n)$

Ganz besonders tricky geht es in Zeit $\mathcal{O}^*(1, 34^n)$, siehe:

> Robin A. Moser, Dominik Scheder. A Full Derandomization of Schöning's k-SAT Algorithm. arXiv:1008.4067, 2010.

Laufzeitvorteil gegenüber A1 entspricht dem Quotienten

- $2^n/1,84^n$ bei A2
- $2^n/1,77^n$ bei A3
- $2^n/1,34^n$ bei tricky



Übersicht

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit n Knoten.

Brute-Force-Ansatz: Suche ein Vertex Cover, indem alle $\binom{n}{k}$ vielen Teilmengen der Größe k betrachtet werden.

Laufzeit: $\mathcal{O}(\binom{n}{k} \cdot |G|) \subseteq \mathcal{O}(n^k \cdot |G|)$

$$\begin{aligned} \binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!} \\ &\leq \frac{n \cdot n \cdot \ldots \cdot n}{\cancel{k!}} \leq n^k \end{aligned}$$

Die Laufzeit ist polynomiell, falls k konstant ist, also nicht zur Eingabe des Problems gehört.

Greedy-Heuristik:

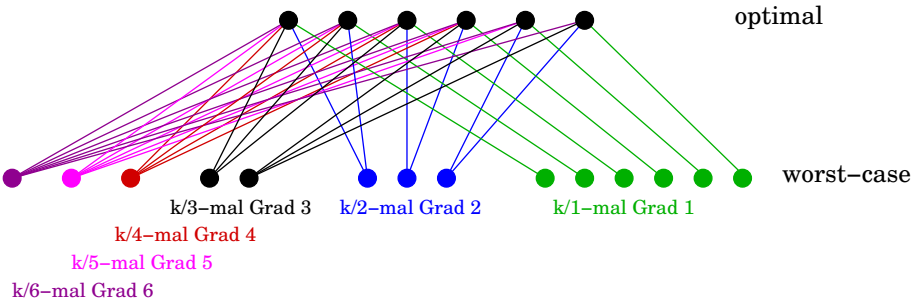
$C := \emptyset$
while es gibt noch Kanten in G **do**
    sei v ein Knoten mit größter Anzahl Nachbarn
    nimm v in C auf
    entferne v und alle inzidenten Kanten aus G

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

Bemerkung: Es gibt Graphen G, für die die Greedy-Heuristik ein Vertex Cover der Größe $\ln(k) \cdot k$ liefert, wobei k die Größe des kleinsten Vertex Cover ist.

Der Fehler kann also $\ln(k)$ groß werden und ist nicht durch eine Konstante beschränkt.

Worst-Case-Graph mit $k = 6$ für die Greedy-Heuristik:



- obiger bipartiter Graph hat ein Vertex Cover der Größe $k = 6$
- im schlechtesten Fall werden die unteren Knoten ins VC aufgenommen, das die Größe $6 + 3 + 2 + 1 + 1 + 1 = 14$ hat.
- Fehler im Beispiel: $\frac{14}{6} \approx 2,33$
- harmonische Reihe: $\lfloor \frac{k}{1} \rfloor + \lfloor \frac{k}{2} \rfloor + \lfloor \frac{k}{3} \rfloor + ... + \lfloor \frac{k}{k} \rfloor = k \cdot \sum_{i=1}^{k} \lfloor \frac{1}{k} \rfloor \approx k \cdot \ln(k)$

Approximationsalgorithmus:

$C := \emptyset$
while es gibt noch Kanten in G **do**
  nimm irgendeine Kante $\{u, v\}$ von G
  nimm u und v beide in C auf
  entferne u und v und alle dazu inzidenten Kanten aus G

Bemerkung: Der Algorithmus liefert für alle Graphen ein VC, dass höchstens doppelt so viele Knoten enthält wie ein minimales VC:

- Sei F die Menge der ausgewählten Kanten, und sei $C = \{u, v \mid \{u, v\} \in F\}$ das berechnete Vertex Cover.
- Jedes Vertex Cover C' muss u oder v enthalten, da sonst die Kante $\{u, v\}$ nicht abgedeckt würde. Also gilt:

$$|C'| \geq \frac{1}{2}|C| \iff |C| \leq 2|C'|$$

Algorithmen für Vertex Cover

Algorithmus binärer Entscheidungsbaum mit beschränkter Tiefe k:

function $\mathrm{VC}(G, k)$
  if $k = 0$ und es gibt noch Kanten in G **then return** false
  if es gibt keine Kanten in G **then return** true
  nimm irgendeine Kante $\{u, v\}$ von G
  $x := \mathrm{VC}(G - \{u\}, k - 1)$
  $y := \mathrm{VC}(G - \{v\}, k - 1)$
  return $x \vee y$

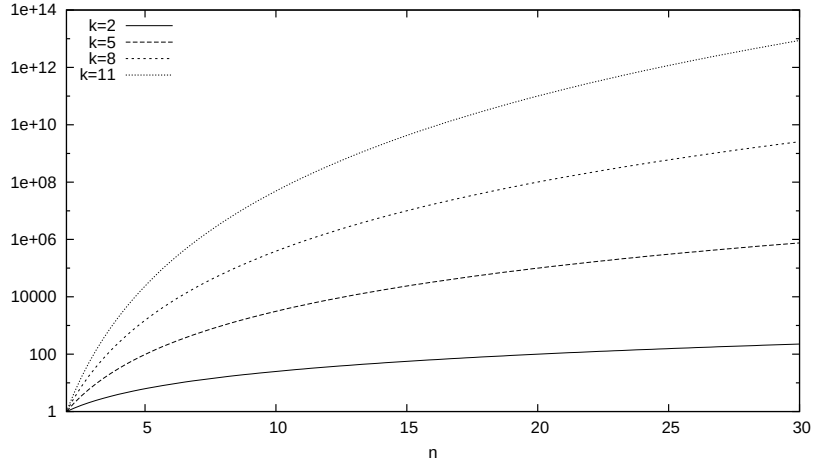
Laufzeit: $\mathcal{O}(2^k \cdot |G|)$

Bemerkungen: Falls k konstant ist

- ergibt sich eine polynomielle Laufzeit und
- die asymptotische Laufzeit hängt nicht von k ab.

Laufzeitvorteil entspricht dem Quotienten $n^k/2^k$

- Brute-Force: $\mathcal{O}(n^k \cdot |G|)$
- Entscheidungsbaum: $\mathcal{O}(2^k \cdot |G|)$



Fixed Parameter Tractable (FPT)

Ein *parametrisiertes Problem* ist ein Paar (Π, k), wobei Π ein Entscheidungsproblem ist, $\mathcal{I}$ die Menge der möglichen Eingaben für Π ist, und $k : \mathcal{I} \to \mathbb{N}$ eine Parametrisierung ist, die sich in polynomieller Zeit berechnen lässt. Der Wert der Parametrisierung sollte bei praktischen Anwendungen klein sein.

Ein Algorithmus A ist ein *fpt-Algorithmus*, wenn die Laufzeit für jede Eingabe $I \in \mathcal{I}$ in $\mathcal{O}(f(k(I)) \cdot |I|^c)$ liegt, wobei f eine Funktion und $c \in \mathbb{N}$ konstant ist.

Ein parametrisiertes Problem (Π, k) gehört zur Klasse *FPT* und wird *fixed parameter tractable* genannt, wenn es einen fpt-Algorithmus bezüglich k gibt, der Π entscheidet.

Vertex Cover ist in FPT bezüglich des Standardparameters k.

Die *Idee bei parametrisierten Algorithmen:* Beschränke die kombinatorische Explosion auf den Parameter k.

Fixed Parameter Tractable (FPT)

Kleine Parameter k finden sich in vielen Anwendungsbereichen:

- Netzwerke: Positioniere eine kleine Anzahl k von Geräten wie Router in einem großen Netzwerk.
- VLSI-Entwurf: Layout in der Chip-Produktion ist beschränkt auf $k < 30$ Verdrahtungsschichten.
- Algorithmische Biologie: Untersuchungen zu DNS-Sequenzen der Länge n für k wenige Spezies.

Ein Problem Π kann bezüglich eines Parameters k_1 in FPT sein, und bezüglich eines anderen Parameters k_2 nicht.

Frage: Lässt sich für Vertex Cover die Laufzeit noch weiter reduzieren?

Die Verkleinerung von Suchbaumgrößen ist wichtig, wie wir bereits bei dem 3Knf-Sat-Problem gesehen haben.

Kernbildung: Reduziere die Eingabe durch Anwendung verschiedener Regeln auf einen schweren, aber kleinen Problemkern.

- Reduziere die Eingabe (I, k) in Zeit $\mathcal{O}(p(|I|))$ auf eine äquivalente Eingabe I', so dass die Größe von I' nur von k und nicht von $|I|$ abhängt, und p ein Polynom ist.
- Löse die äquivalente Eingabe I' mit erschöpfender Suche. Die Laufzeit für diesen Schritt hängt dann nur noch von k, aber nicht von $|I|$ ab.

Beobachtungen für ein Vertex Cover $V' \subseteq V$ für einen Graphen $G = (V, E)$ mit $|V'| \leq k$:

- Für einen Knoten $v \in V$ liegt v selbst oder seine gesamte Nachbarschaft $N(v)$ im Vertex Cover V'.
- V' enthält jeden Knoten $v \in V$ mit Knotengrad $deg(v) > k$. Denn würde v nicht ins Vertex Cover V' aufgenommen, dann müssten mehr als k viele Nachbarn von v im Vertex Cover liegen. $\lightning$

Jedes Vertex Cover für G hat mehr als k viele Knoten, falls

- $\Delta(G) \leq k$ ist, also jeder Knoten im Graphen G einen Knotengrad höchstens k hat, **und**
- $|E| > k^2$ ist, also mehr als k^2 viele Kanten im Graphen enthalten sind,

denn jeder der k Knoten kann höchstens k viele Kanten abdecken.

Sei $G = (V, E)$ ein Graph und $k \in \mathbb{N}$. Der folgende Algorithmus basiert auf den obigen Beobachtungen.

function VertexCover(G, k)

$H := \{v \in V \mid deg(v) > k\}$	$\mathcal{O}(n)$
if $\lvert H\rvert > k$ **then return** false	$\mathcal{O}(1)$
$k' := k - \lvert H\rvert$	$\mathcal{O}(1)$
$G' := G - H$	$\mathcal{O}(n \cdot k)$
if $\lvert E(G')\rvert > k \cdot k'$ **then return** false	$\mathcal{O}(n)$
entferne alle isolierten Knoten aus G'	$\mathcal{O}(n)$
berechne ein Vertex Cover der Größe k' in G'	$\mathcal{O}(2^k \cdot k^2)$

Laufzeit: $\mathcal{O}(n \cdot k + 2^k \cdot k^2)$ mit $n = \lvert V\rvert$ und $k' \leq k$.

Frage: Wie wird der Graph gespeichert, und wie werden die einzelnen Operationen ausgeführt, um die angegebenen Laufzeiten zu erreichen?

Wir speichern den Graphen mittels *Adjazenzlisten*: Zu jedem Knoten wird eine Liste der Nachbarknoten gespeichert.

- Jede Liste hat Länge $\mathcal{O}(n)$, da jeder Knoten höchstens mit $n-1$ anderen Knoten durch eine Kante verbunden sein kann.
- Die Länge einer Liste wird in einer Variablen gespeichert und kann daher in Zeit $\mathcal{O}(1)$ ermittelt werden: Beim Einfügen in oder Entfernen aus der Liste aktualisiere diese Variable.

Bleibt noch zu klären, wie $G' := G - H$ berechnet wird:

- Lösche für jeden Knoten $v \in H$ seine Nachbarschaftsliste.
  $\Rightarrow$ Übrig bleibende Listen haben Länge höchstens k.
- Durchlaufe jede verbliebene Liste und lösche Knoten $v \in H$.
  $\Rightarrow$ Höchstens n Listen der Länge höchstens k müssen durchlaufen werden und Elemente daraus gelöscht werden: $\mathcal{O}(n \cdot k)$

Die Anzahl der zu lösenden Teilprobleme und damit die Laufzeit hängt von der Struktur des Baumes ab:

- Binärbaum: $T(k) \leq T(k - t_1) + T(k - t_2) + c$
- allgemein: $T(k) \leq T(k - t_1) + \ldots + T(k - t_s) + c$

Der Vektor $(t_1, \ldots, t_s)$ heißt *Verzweigungsvektor*. Dabei geben $t_1, \ldots, t_s$ an, um wie viel kleiner die Teilprobleme im Vergleich zum ursprünglichen Problem sind.

Beispiel: Ein Binärbaum, bei dem in beiden Zweigen das zu lösende Teilproblem um eins kleiner ist als im ursprünglichen Problem, wenn also $t_1 = 1$ und $t_2 = 1$ gilt, hat den Verzweigungsvektor $(1, 1)$. (z.B. Algorithmus A1 für 3KNF-SAT)

Die Zahl der Teilprobleme ist exponentiell in k, also $T(k) \in \mathcal{O}(b^k)$, wobei die Basis b vom Verzweigungsvektor abhängt:

Vektor	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(3,3)	(3,3,6)	(3,4,6)
Basis b	2	1,618	1,466	1,381	1,325	1,325	1,26	1,342	1,305

Gedanken-Experiment für Vertex Cover:

Angenommen, es gibt nach der Kernbildung immer einen Knoten v mit $deg(v) \geq 4$.

- Wähle in jedem Schritt entweder v oder alle seine Nachbarn und verzweige im binären Entscheidungsbaum entsprechend.
  $\Rightarrow$ Verzweigungsvektor $(1, 4)$
- Laufzeit: $\mathcal{O}(n \cdot k + 1,381^k \cdot k^2)$

Problem: Es gibt nicht in jedem Schritt einen Knoten mit Grad mindestens vier.

Idee: Verzweige in jedem Schritt je nach eintretendem Fall mit Verzweigungsvektor $(1, 5)$, $(2, 3)$, $(3, 3)$, $(3, 4, 6)$ oder $(3, 3, 6)$ oder höchstens einmal pro Pfad $(1, 4)$.

Gesamtlaufzeit: $\mathcal{O}(n \cdot k + 1,342^k \cdot k^2)$

Verzweige anhand der Bedingung mit der kleinsten Nummer:

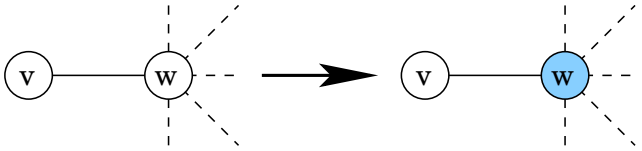
1. falls ein Knoten v mit $deg(v) = 1$ existiert $\rightsquigarrow$ Regel 1
2. falls ein Knoten v mit $deg(v) \geq 5$ existiert $\rightsquigarrow$ Regel 2
3. falls ein Knoten v mit $deg(v) = 2$ existiert $\rightsquigarrow$ Regel 3
4. falls ein Knoten v mit $deg(v) = 3$ existiert $\rightsquigarrow$ Regel 4
5. falls alle Knoten Grad 4 haben $\rightsquigarrow$ Regel 5

Offensichtlich ist immer eine der obigen Bedingungen erfüllt.

Auf den folgenden Folien beschreiben wir die Regeln im Detail.

Regel 1: Es existiert ein Knoten v mit $deg(v) = 1$.

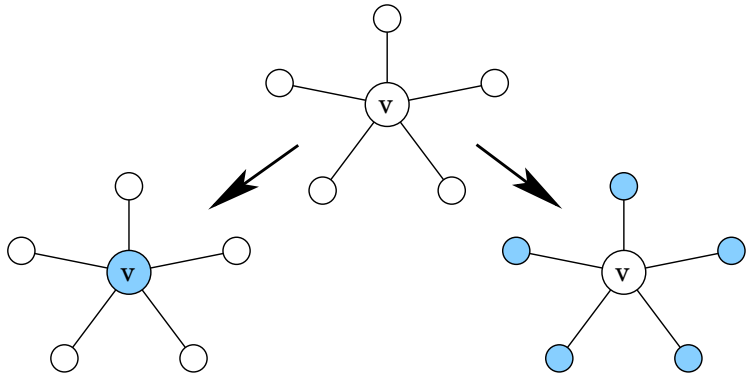
Sei w der Nachbar von v.



$\Rightarrow$ Es kann nie besser sein, v statt w zu nehmen.

$\Rightarrow$ Keine Verzweigung nötig: Nimm w ins Vertex Cover auf!

Sicher: alle Knoten haben Grad $2, 3, 4, 5, \ldots$
Regel 2: Es existiert ein Knoten v mit $deg(v) \geq 5$.

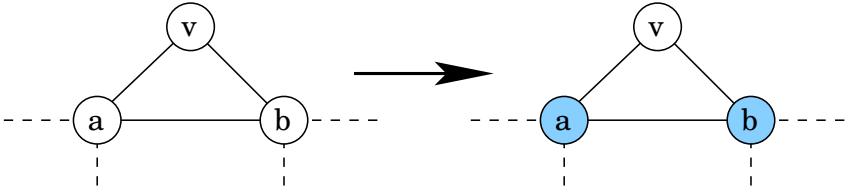


$\Rightarrow$ Wähle entweder v oder alle Nachbarn von v.

$\Rightarrow$ Verzweigungsvektor: $(1, 5)$

Sicher: alle Knoten haben Grad $2, 3$ oder 4
Regel 3: Es existiert ein Knoten v mit $deg(v) = 2$.

Fall 1: Die Nachbarn a und b von v sind adjazent.



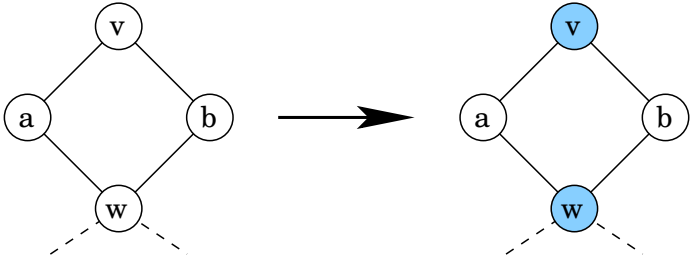
$\Rightarrow$ Zwei der Knoten v, a, b müssen gewählt werden

$\Rightarrow$ Es ist nie besser v zu wählen.

$\Rightarrow$ Keine Verzweigung nötig: Nimm a und b ins Vertex Cover auf!

Sicher: alle Knoten haben Grad $2, 3$ oder 4
Regel 3: Es existiert ein Knoten v mit $deg(v) = 2$.

Fall 2: Die Nachbarn a und b von v haben Grad zwei und einen gemeinsamen Nachbarn w.



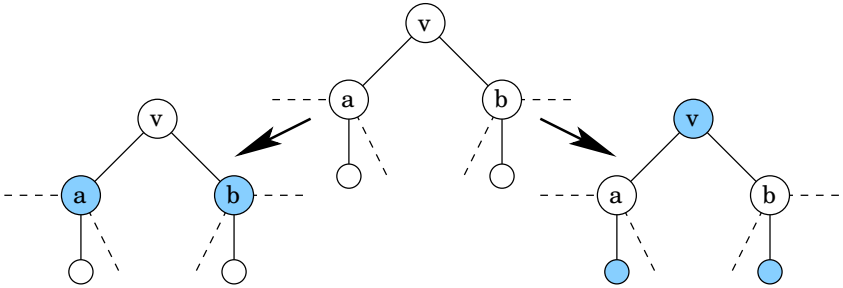
$\Rightarrow$ Zwei der Knoten v, a, b, w müssen gewählt werden.

$\Rightarrow$ Es ist nie besser a und b zu wählen.

$\Rightarrow$ Keine Verzweigung: Nimm v und w ins Vertex Cover auf!

Sicher: alle Knoten haben Grad $2, 3$ oder 4
Regel 3: Es existiert ein Knoten v mit $deg(v) = 2$.

Fall 3: sonst (a und b sind nicht adjazent, a oder b haben Grad größer als zwei oder haben keinen gemeinsamen Nachbarn)



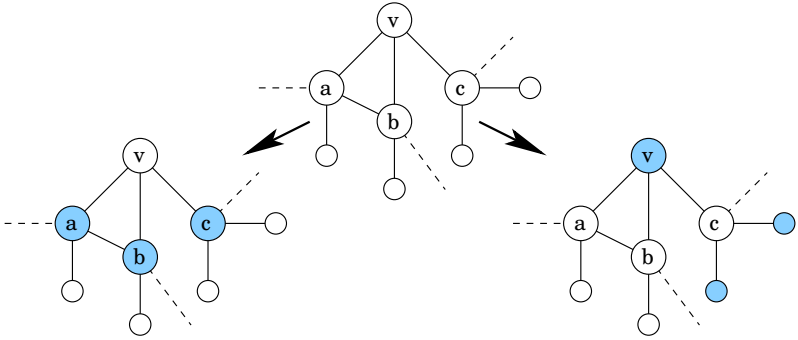
$\Rightarrow$ Falls a oder b gewählt wurde: sinnlos v zu wählen

$\Rightarrow$ Wähle entweder a und b, oder die Nachbarn von a und b.

$\Rightarrow$ Verzweigungsvektor: $(2, 3)$

Sicher: alle Knoten haben Grad 3 oder 4
Regel 4: Es existiert ein Knoten v mit $deg(v) = 3$.

Fall 1: v ist Teil eines Dreiecks aus v, a und b. Mindestens zwei der Knoten müssen in einem VC liegen.

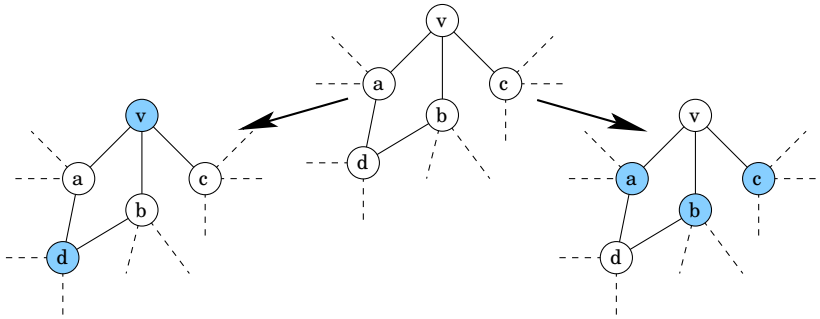


$\Rightarrow$ Falls c gewählt wurde, macht es keinen Sinn, v zu wählen.
$\Rightarrow$ Füge die Nachbarn von v oder die Nachbarn von c hinzu.
$\Rightarrow$ Verzweigungsvektor: $(3, 3)$

Sicher: alle Knoten haben Grad 3 oder 4

Regel 4: Es existiert ein Knoten v mit $deg(v) = 3$.

Fall 2: v ist Teil eines Vierecks aus v, a, d und b.



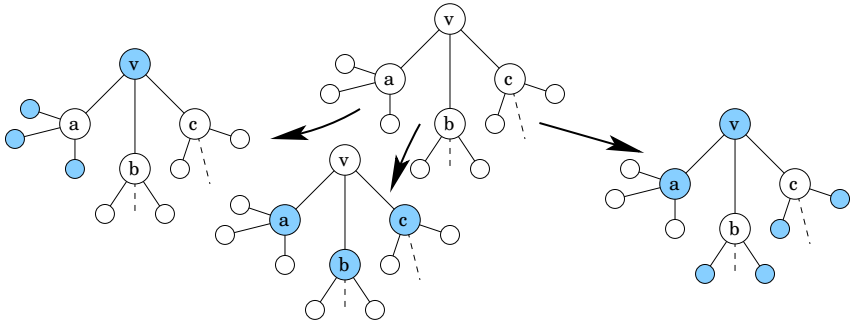
$\Rightarrow$ In jedem Vertex Cover sind zumindest v und d oder a und b enthalten. Wenn a und b enthalten sind: sinnlos v zu wählen.

$\Rightarrow$ Füge a, b und c oder v und d hinzu.

$\Rightarrow$ Verzweigungsvektor: $(2, 3)$

Sicher: alle Knoten haben Grad 3 oder 4
Regel 4: Es existiert ein Knoten v mit $deg(v) = 3$.

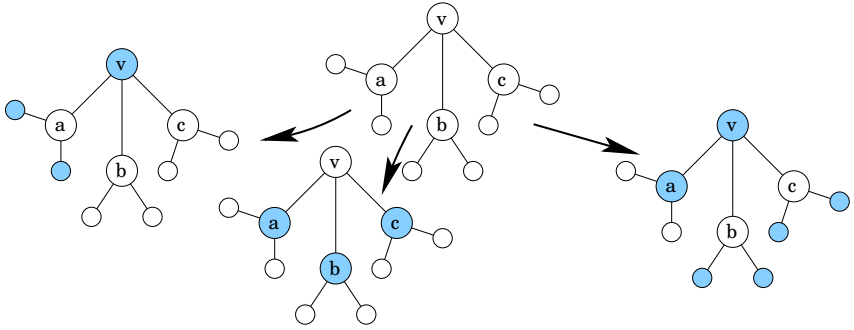
Fall 3: v ist in keinem Dreieck oder Viereck enthalten
Fall a: $deg(a) = 4$ (analog für b oder c)



$\Rightarrow$ Ein Vertex Cover enthält a oder alle Nachbarn von a.
$\Rightarrow$ Ist a und (b oder c) enthalten: sinnlos v zu wählen
$\Rightarrow$ Verzweigungsvektor: $(3, 4, 6)$

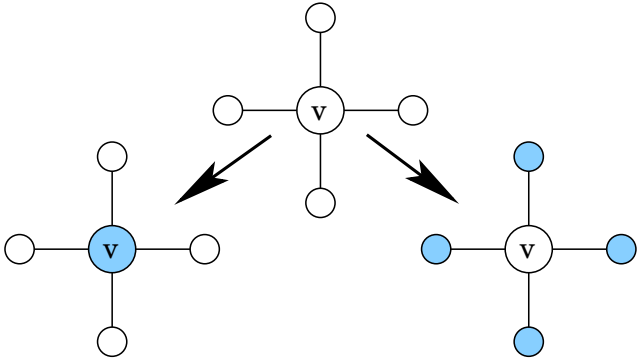
Sicher: alle Knoten haben Grad 3 oder 4
Regel 4: Es existiert ein Knoten v mit $deg(v) = 3$.

Fall 3: v ist in keinem Dreieck oder Viereck enthalten
Fall b: $deg(a) = deg(b) = deg(c) = 3$ (sonst)



$\Rightarrow$ Ein Vertex Cover enthält a oder alle Nachbarn von a.
$\Rightarrow$ Ist a und (b oder c) enthalten: sinnlos v zu wählen
$\Rightarrow$ Verzweigungsvektor: $(3, 3, 6)$

Regel 5: Alle Knoten haben Grad vier.



- Wähle entweder v oder alle seine Nachbarn.

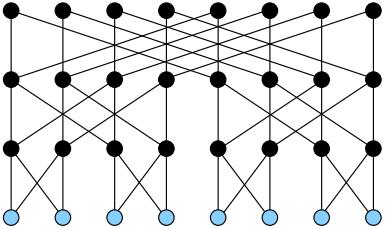
$\Rightarrow$ Verzweigungsvektor $(1, 4)$

- Danach enthält jeder Subgraph mindestens einen Knoten mit kleinerem Grad!

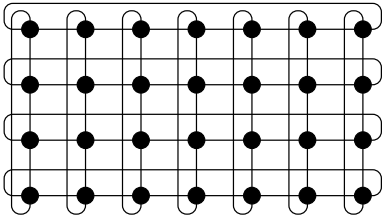
$\Rightarrow$ Die Regel wird pro Pfad nur einmal ausgeführt.

Es gibt reguläre Graphen vom Grad 4:

Wrapped Butterfly Network:

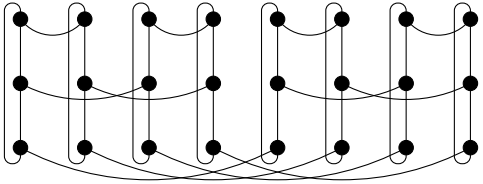


Torus:

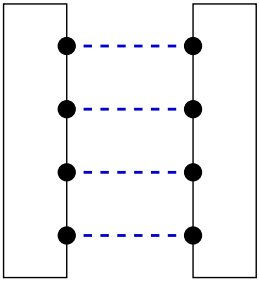


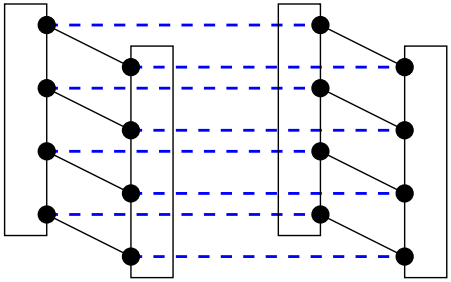
Knoten der oberen und unteren Reihe des Butterfly sind identisch.

Regulärer Graph vom Grad 3: Cube Connected Cycles



Aus zwei Kopien eines k-regulären Graphen kann ein $k+1$-regulärer Graph erstellt werden:





Integer Programming for Vertex Cover

minimize

$$\sum_{v \in V} x_v$$

subject to

$$x_u + x_v \geq 1 \quad \forall \{u, v\} \in E$$
$$x_u \in \{0, 1\} \quad \forall u \in V$$

Übersicht

Tiefenbeschränkte Suchbäume

- sind nicht immer zum Lösen von schweren Problemen geeignet
- und sind bei Maximierungsproblemen oft ungeeignet.

Bei Maximum Independent Set ist eine möglichst große Teilmenge gesucht:

- Der Suchbaum müsste die Tiefe n haben.
- Das würde aber zu einer großen Laufzeit führen.

Sei v ein beliebiger Knoten. Dann sei $\hat{N}(v)$ die Menge der Nachbarn von v inklusive v selbst.

Algorithmus B1: Falls ein Knoten v ins I.S. aufgenommen wird, dann sind alle Nachbarn von v nach Definition nicht im I.S.

function MaxIndependentSet($G = (V, E)$)
  if $E = \emptyset$ **then**
    return $|V|$
  let v be some arbitrary node in G
  $a := 1 +$ MaxIndependentSet($G - \hat{N}(v)$)
  $b :=$ MaxIndependentSet($G - \{v\}$)
  return $\max\{a, b\}$

Laufzeit: $T(n) = 2 \cdot T(n-1) + p(n) \ \in \ \mathcal{O}^\star(2^n)$

Obige Laufzeit ergibt sich, falls v keinen Nachbarn hat, also $\hat{N}(v) = \{v\}$ ist. Dann sind aber beide rekursiven Aufrufe gleich.

Betrachten wir diesen Worst-Case genauer:

- Wenn $\hat{N}(v) = \{v\}$ ist, also v keine Nachbarn hat, dann gehört v zu jedem Maximum Independent Set und wir können auf den zweiten rekursiven Aufruf verzichten.
- Ansonsten hat v mindestens einen Nachbarn, und $G - \hat{N}(v)$ hat höchstens $n - 2$ Knoten.

Wir können daher genauer abschätzen, mit unseren guten, alten Freunden, den Fibonacci-Zahlen:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-1) + T(n-2) \end{array} \right\} + p(n) \ \in \ \mathcal{O}^\star(1,6181^n)$$

Neuer Worst-Case: v hat genau einen Nachbarn w.

- Entweder ist v oder w im Independent Set, aber nicht beide.
- Falls ein Independent Set w enthält, dann gibt es ein ebenso großes Independent Set, das statt w den Knoten v enthält.
- Wir können also davon ausgehen, dass es ein Maximum Independent Set gibt, das den Knoten v enthält.
- Daher muss der Graph $G - \{v\}$ gar nicht untersucht werden, es reicht aus, den Graphen $G - \hat{N}(v)$ zu untersuchen, der höchstens $n-2$ Knoten hat.

Falls v mehr als einen Nachbarn hat, dann hat $G - \hat{N}(v)$ höchstens $n-3$ Knoten, und wir schätzen ab:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-3) \end{array} \right\} + p(n) \ \in \ \mathcal{O}^\star(1,4658^n)$$

Neuer Worst-Case: Jeder Knoten hat Grad mindestens 2.

- Falls G einen Knoten v vom Grad mindestens drei hat, dann hat $G - \hat{N}(v)$ höchstens $n - 4$ Knoten.
- Sonst hat jeder Knoten den Grad zwei. Sei u, v, w ein Weg in G. Dann gilt für jedes maximale Independent Set:
  - Entweder v ist enthalten, aber u und w nicht, oder
  - u ist enthalten, aber seine beiden Nachbarn nicht, oder
  - w ist enthalten, aber seine beiden Nachbarn nicht.

  In allen drei Fällen erhalten wir rekursive Aufrufe mit höchstens $n - 3$ Knoten.

Laufzeit:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \\ 3 \cdot T(n-3) \end{array} \right\} + p(n) \ \in \ \mathcal{O}^\star(1,4423^n)$$

Obige Laufzeit wird bestimmt durch den Worst-Case, wo jeder Knoten Grad 2 hat. Schauen wir uns diesen Spezialfall genauer an:

- Wenn jeder Knoten den Grad 2 hat, dann besteht jede Komponente des Graphen aus einem Kreis.
- Ein Kreis der Länge k hat maximal ein Independent Set der Größe $\lfloor k/2 \rfloor$.
- Es ist also gar kein rekursiver Aufruf nötig: Dieser Spezialfall wird direkt in polynomieller Zeit gelöst.

Daher ergibt sich als Laufzeit:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \end{array} \right\} + p(n) \ \in \ \mathcal{O}^\star(1,3803^n)$$

Algorithmen für Independent Set

function MaxIndependentSet($G = (V, E)$)
  if $E = \emptyset$ **then**
    return $|V|$
  else if G hat Knoten v mit Grad 0 oder 1 **then**
    return $1 +$ MaxIndependentSet($G - \hat{N}(v)$)
  else if G hat Knoten v mit Grad größer als 2 **then**
    $a := 1 +$ MaxIndependentSet($G - \hat{N}(v)$)
    $b :=$ MaxIndependentSet($G - \{v\}$)
    return $\max\{a, b\}$
  else
    $total := 0$
    for all Komponente K von G **do**
      $total := total + \lfloor |K|/2 \rfloor$
    return $total$

Es gibt weitere Verbesserungen, die aber zunehmend komplex sind:

> Fomin, Grandoni, Kratsch. Measure and Conquer: A simple $\mathcal{O}(2^{0{,}288n})$ independent set algorithm. Proceedings of SODA, 18 – 25, 2006.
>
> Anmerkung: $2^{0{,}288n} = 1.2209465^n$

Integer Programming for Independent Set

maximize

$$\sum_{v \in V} x_v$$

subject to

$$\begin{aligned} x_u + x_v &\leq 1 \quad \forall \{u, v\} \in E \\ x_u &\in \{0, 1\} \quad \forall u \in V \end{aligned}$$

Übersicht

Übersicht

Divide and Conquer

Entwurfsprinzip:

- *Divide* the problem into subproblems.
- *Conquer* the subproblems by solving them recursively.
- *Combine* subproblem solutions.

Beispiele:

- Binäre Suche
- Potenzieren einer Zahl
- Matrix-Multiplikation
- Quicksort

Potenzieren einer Zahl

Problem: Berechne x^n für ein $n \in \mathbb{N}$.

- *Einfacher Algorithmus:*

```
erg := 1
for i := 1 to n do
    erg := erg * x
```

  $\rightarrow$ Laufzeit: $\Theta(n)$ Multiplikationen

- *Divide & Conquer:*

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{falls } n \text{ gerade} \\ x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x & \text{sonst} \end{cases}$$

  Der rekursive Aufruf erfolgt natürlich nur einmal.

Potenzieren einer Zahl

Divide & Conquer:

function POWER(x: real, n: int) : real
  if $n = 1$ **then**
    return x
  if n is even **then**
    $t := \text{POWER}(x, n/2)$
    return $t \cdot t$
  else
    $t := \text{POWER}(x, (n-1)/2)$
    return $t \cdot t \cdot x$

$\rightarrow$ Laufzeit: $T(n) \leq T(n/2) + c \in \mathcal{O}(???)$

Induktive Einsetzungsmethode

- Rate eine Lösung und bestätige diese durch vollst. Induktion.
- *Beispiel:* Für $T(n) = 2 \cdot T(n/2) + b$ "raten" wir $T(n) \leq c \cdot n - a$ als Lösung und rechnen nach:

$$\begin{aligned} T(n) = 2 \cdot T(n/2) + b &\leq 2 \cdot (c \cdot \frac{n}{2} - a) + b \\ &\leq c \cdot n - 2a + b \\ &= c \cdot n - a + (b - a) \\ &\leq c \cdot n - a \text{ für } b \leq a \end{aligned}$$

- Problem ist falsches Raten. Nehmen wir bei obigem Beispiel $T(n) \in \mathcal{O}(n)$ an, also $T(n) \leq c \cdot n$, dann erhalten wir:

$$\begin{aligned} T(n) = 2 \cdot T(n/2) + b &\leq 2 \cdot (c \cdot \frac{n}{2}) + b \\ &\leq c \cdot n + b \; \lightning \end{aligned}$$

Iterative Methode

- Setze Rekursionsgleichung fort bis zu einer geschlossenen Form.
- *Beispiel:* $T(n) = 2 \cdot T(n/2) + b$

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + b \\ &= 2 \cdot (2 \cdot T(n/4) + b) + b = 4 \cdot T(n/4) + 3b \\ &= 4 \cdot (2 \cdot T(n/8) + b) + 3b = 8 \cdot T(n/8) + 7b \\ &\vdots \\ &= 2^k \cdot T(n/2^k) + (2^k - 1) \cdot b \end{aligned}$$

Für $n = 2^k \iff \log_2(n) = k$ und $T(1) = const$ erhalten wir:

$$T(n) = n \cdot T(1) + (n-1) \cdot b \in \mathcal{O}(n)$$

Variablensubstitution

- Ersetze n durch einen Ausdruck, so dass die Rekursionsgleichung eine bekannte Form bekommt.
- *Beispiel:* Für $T(n) = 2 \cdot T(\sqrt{n}) + \log_2(n)$ mit der Substitution $m = \log_2(n) \iff 2^m = n$ erhalten wir

$$\begin{aligned} T(2^m) &= 2 \cdot T(\sqrt{2^m}) + m \\ &= 2 \cdot T(2^{\frac{m}{2}}) + m \end{aligned}$$

Löse Gleichung mit einer der ersten beiden Methoden.

Eingabe: zwei $n \times n$-Matrizen A und B
Ausgabe: $C = A \cdot B$

Es gilt:

$$\begin{pmatrix} c_{11} & \cdots & c_{1n} \\ c_{21} & \cdots & c_{2n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}$$

mit

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

Einfacher Algorithmus:

```
for i := 1 to n do
    for j := 1 to n do
        c[i][j] := 0
        for k := 1 to n do
            c[i][j] := c[i][j] + a[i][k] * b[k][j]
```

$\rightarrow$ Laufzeit: $\Theta(n^3)$ Additionen/Multiplikationen

Die Laufzeit ergibt sich aus der Tabellengröße mal Aufwand pro Eintrag: Die Tabelle hat n^2 Einträge, für jeden Eintrag wird eine Summe über n viele Produkte gebildet.

Aufteilen der $n \times n$-Matrizen in jeweils vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen:

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

$$C = A \cdot B$$

mit

$$\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}$$

$\Rightarrow$ 8 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen
4 Additionen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen

$\rightarrow$ Laufzeit: $T(n) = 8 \cdot T(n/2) + 4 \cdot (n/2)^2 = \ldots \in \Theta(n^3)$

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

mit

$$\begin{aligned} P_1 &= a \cdot (f - h) \\ P_2 &= (a + b) \cdot h \\ P_3 &= (c + d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{aligned}$$

und

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$\rightarrow$ Laufzeit: $T(n) = 7\,T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2{,}807})$

Vergleich der Laufzeiten unter der Annahme, dass ein Rechner $20 \cdot 10^9$ charakteristische Operationen pro Sekunde ausführen kann:

n	n^3	time	$n^{2,807}$	time	$n^{2,397}$	time
1 e3	1 e9	<1 s	2,64 e8	<1 s	1,55 e7	<1 s
10 e3	1 e12	50 s	1,69 e11	9 s	3,87 e9	<1 s
100 e3	1 e15	14 h	1,08 e14	90 m	9,66 e11	49 s
1.000 e3	1 e18	2 j	6,95 e16	40 t	2,41 e14	201 m
10.000 e3	1 e21	1.585 j	4,46 e19	71 j	6,01 e16	35 t

s: Sekunden, m: Minuten, h: Stunden, t: Tage, j: Jahre

Beachte: Obige Laufzeitabschätzungen zur Matrixmultiplikation zählen nur die Anzahl der Multiplikationen und Additionen. Wir berücksichtigen dabei nicht die Größe der Zahlen!

- Addition zweier Zahlen der Länge ℓ nach Schulmethode: $\mathcal{O}(\ell)$
- Multiplikation zweier Zahlen der Länge ℓ nach
  - Schulmethode: $\mathcal{O}(\ell^2)$
  - Karazuba: $\mathcal{O}(\ell^{\log_2(3)}) = \mathcal{O}(\ell^{1,5849})$
  - Schönhage/Strassen: $\mathcal{O}(\ell \cdot \log(\ell) \cdot \log(\log(\ell)))$

Die Laufzeit der einfachen Matrix-Multiplikation unter Berücksichtigung der Größe der Zahlen ergäbe nach

- Schulmethode: $\mathcal{O}(n^3 \cdot \ell^2)$
- Karazuba: $\mathcal{O}(n^3 \cdot \ell^{\log_2(3)})$
- Schönhage/Strassen: $\mathcal{O}(n^3 \cdot \ell \cdot \log(\ell) \cdot \log(\log(\ell)))$

Die Zifferntupel seien $X = (x_{2n-1} \dots x_0)_b$ und $Y = (y_{2n-1} \dots y_0)_b$.

Jedes Zifferntupel aufspalten in zwei Tupel der Länge n:

$$X_h = (x_{2n-1} \dots x_n)_b \quad \textit{und} \quad X_l = (x_{n-1} \dots x_0)_b$$
$$Y_h = (y_{2n-1} \dots y_n)_b \quad \textit{und} \quad Y_l = (y_{n-1} \dots y_0)_b.$$

Damit ist $X = X_h b^n + X_l$ und $Y = Y_h b^n + Y_l$ und wir erhalten

$$XY = X_h Y_h b^{2n} + (X_h Y_l + X_l Y_h)\, b^n + X_l Y_l.$$

Den Term $X_h Y_l + X_l Y_h$ in andere Form bringen:

$$\begin{aligned} X_h Y_l + X_l Y_h &= (X_h Y_h + X_h Y_l + X_l Y_h + X_l Y_l) - (X_h Y_h + X_l Y_l) \\ &= (X_h + X_l)(Y_h + Y_l) - (X_h Y_h + X_l Y_l). \end{aligned}$$

Dann sind im Produkt nur noch drei Produkte enthalten:

$$XY = X_h Y_h b^{2n} + ((X_h + X_l)(Y_h + Y_l) - (X_h Y_h + X_l Y_l))\, b^n + X_l Y_l$$

Master-Theorem:

$$T(n) = a \cdot T(n/b) + \Theta(n^k) \quad \text{mit} \quad T(1) \in \mathcal{O}(1)$$

Unterscheide drei Fälle:

- für $a < b^k$ gilt: $T(n) \in \Theta(n^k)$
- für $a = b^k$ gilt: $T(n) \in \Theta(n^k \cdot \log(n))$
- für $a > b^k$ gilt: $T(n) \in \Theta(n^{\log_b a})$

Herleitung: $\rightarrow$ siehe Vorlesung ALD im Bachelor-Studium

Beispiele:

- *Binäre Suche:* $a = 1,\ b = 2,\ k = 0$

$$a = b^k \ \rightarrow \ T(n) = \Theta(n^k \cdot \log(n)) = \Theta(\log(n))$$

- *Matrix-Multiplikation:* $a = 8,\ b = 2,\ k = 2$

$$a > b^k \ \rightarrow \ T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

- *Merge-Sort:* $a = 2,\ b = 2,\ k = 1$

$$a = b^k \ \rightarrow \ T(n) = \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

Gegeben: Ein Array a mit n ganzen Zahlen

Gesucht: Zwei Indizes i und j, sodass die Summe $a[i] + \ldots + a[j]$ möglichst groß ist.

naiv: Teste alle $\binom{n}{2}$ Paare (i, j) mit $1 \leq i < j \leq n$ und wähle das Paar mit größter Summe $a[i] + \ldots + a[j]$. Laufzeit ist dann in $\Omega(n^2)$.

Übung 21. *Entwickeln Sie einen Divide-and-Conquer-Algorithmus und schätzen Sie die Laufzeit des Algorithmus ab.*

Übersicht

Greedy- (gierige) Algorithmen sind geeignet, um Optimierungsprobleme zu lösen oder zu approximieren.

Es wird zwischen *exakten Greedy-Algorithmen* und *Greedy-Heuristiken* unterschieden.

Beispiele

- Wechselgeldproblem (exakt oder approximativ)
- Rucksackproblem (exakt oder approximativ)
- kürzeste Wege (exakt)
- minimaler Spannbaum (exakt)
- Strategien für das metrische TSP-Problem (approximativ)

Damit eine optimale Lösung gefunden wird, muss das *Optimalitätsprinzip von Bellman* gelten: „Eine optimale Lösung setzt sich aus optimalen Teillösungen zusammen."

Dies ist allerdings nur eine notwendige Bedingung, keine hinreichende. Die Optimalität muss jeweils explizit bewiesen werden.

Vorgehen bzw. Idee:

- Jeder Schritt wird nur aufgrund der „lokal verfügbaren“ Information durchgeführt.
- Es wird aus allen möglichen „Fortsetzungen einer Teillösung“ diejenige ausgewählt, die momentan den besten Erfolg bringt.

→ Es werden also nicht verschiedene Kombinationen von Teillösungen getestet, sondern nur eine einzige ausgewählt. Diese Lösung ist ggf. nicht optimal.

Korrektheit: Beweise zur Korrektheit setzen sich in der Regel aus zwei Teilen zusammen.

- Es muss gezeigt werden, dass die Lösung zulässig ist, dass also keine Randbedingungen verletzt werden. (Das ist oft der einfachere Teil.)
- Und es muss gezeigt werden, dass die Lösung optimal ist, dass es also keine bessere Lösung gibt. (Es kann aber gleich gute Lösungen geben.)

Die Optimalität wird oft durch ein *Austauschargument* gezeigt. Dies ist speziell dann möglich, wenn die Lösungen die gleiche Größe haben[5] und sich nur in den Kosten unterscheiden können.

- Sei $A(I)$ die Greedy-Lösung zur Eingabe I und sei $O(I)$ eine optimale Lösung.
- Wenn $A(I) \neq O(I)$ ist, dann müssen sich die Lösungen irgendwie unterscheiden. So gilt vielleicht $a \in A(I)$, $a \notin O(I)$ und $o \in O(I)$, $o \notin A(I)$.

Dann ist zu zeigen:

- $O(I) - \{o\} \cup \{a\}$ ist auch eine zulässige Lösung und nicht schlechter als $O(I)$. (Das Element o wird durch das Element a ersetzt. $\rightarrow$ Austausch)
- Die Lösung $O(I)$ kann sukzessiv durch wiederholtes Austauschen in die Lösung $A(I)$ transformiert werden.

Dann ist $A(I)$ nicht schlechter als $O(I)$ und der Greedy-Algorithmus ist daher korrekt.

[5] z.B. haben alle minimalen Spannbäume dieselbe Anzahl Kanten

Optimal Merge Patterns

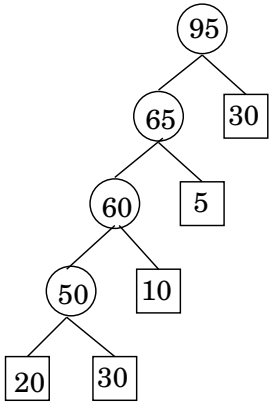
Aus Ellis Horowitz und Sartaj Sahni: Algorithmen – Entwurf und Analyse. Springer Verlag.

- Two sorted files containing q_1 and q_2 records respectively could be merged together to obtain one sorted file in time $\mathcal{O}(q_1 + q_2)$.
- When more than two sorted files are to be merged together the merge can be accomplished by repeatedly merging sorted files in pairs.
- Different pairings require different amounts of computing time.
- *Greedy:* At each step merge the two smallest sized files together.

Optimal Merge Patterns

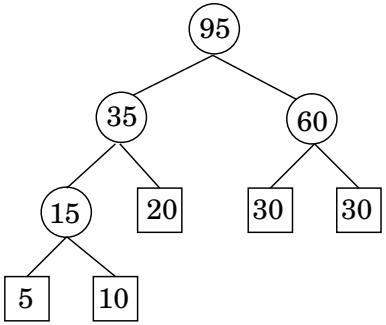
Example: $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$

given order:



cost: $50 + 60 + 65 + 95 = 270$

greedy order:



cost: $15 + 35 + 60 + 95 = 205$

$\rightarrow$ Correctness?

Korrektheit für n Dateien:

- Sei d_i die Entfernung von der Wurzel zum externen Knoten F_i, d.h. die Sätze von F_i werden d_i-mal verschoben.
- Sei q_i die Anzahl der Sätze von F_i, also die Länge von F_i.
- Die gewichtete externe Pfadlänge ist definiert als: $\sum_{i=1}^{n} d_i \cdot q_i$

Induktionsanfang:

- Für $n = 1$ hat der Baum keine inneren Knoten. ✓
- Für $n = 2$ hat der Baum nur einen inneren Knoten. ✓

Induktionsschluss $n \rightsquigarrow n + 1$:

- O.B.d.A. sei $q_1 \leq q_2 \leq \cdots \leq q_n \leq q_{n+1}$
- Im ersten Schritt von Greedy wird F_1 und F_2 gemischt und es entsteht $F_{1/2}$ mit Länge $q_1 + q_2$.

- Sei T ein optimaler Mischbaum für $F_1, F_2, \ldots, F_{n+1}$ und sei v ein innerer Knoten mit maximaler Entfernung zur Wurzel.
- Falls F_1 und F_2 nicht Nachfolger von v in T sind, dann ersetze die Nachfolger F_i und F_j von v durch F_1 und F_2, wodurch die externe Pfadlänge für T nicht vergrößert wird:

$$\begin{aligned} & q_i \cdot d_{max} + q_j \cdot d_{max} + \ldots + q_1 \cdot d_1 + q_2 \cdot d_2 \\ \geq \quad & q_1 \cdot d_{max} + q_2 \cdot d_{max} + \ldots + q_i \cdot d_1 + q_j \cdot d_2 \end{aligned}$$

  Denn:

$$(q_i - q_1) \cdot d_{max} + (q_j - q_2) \cdot d_{max} \;\geq\; (q_i - q_1) \cdot d_1 + (q_j - q_2) \cdot d_2$$

- Dann kann der innere Knoten v durch einen externen Knoten $F_{1/2}$ mit Länge $q_1 + q_2$ ersetzt werden. Der so entstandene Baum ist ein optimaler Mischbaum für $F_{1/2}, F_3, F_4, \ldots, F_{n+1}$.
- Nach Induktionsvoraussetzung liefert Greedy dafür einen optimalen Mischbaum.

Fractional Knapsack Problem

- We are given n objects and a knapsack.
- Object i has weight w_i, and profit p_i.
- The knapsack has a capacity of M.
- If a fraction x_i of object i is placed into the knapsack then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximises the total profit earned.
- *Greedy:* Include next the object which has the maximum profit per unit of capacity used.

Example:

$M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$

Greedy: Include next the object with

1. largest profit. $\rightarrow$ not optimal
2. lowest capacity. $\rightarrow$ not optimal
3. maximum profit per unit of capacity used. $\rightarrow$ Correctness?

Algo	x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	1	$\frac{2}{15}$	0	20	28,2
2	0	$\frac{2}{3}$	1	20	31
3	0	1	$\frac{1}{2}$	20	31,5

Korrektheit für n Objekte: Es gelte $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}$.

- Wir werden zeigen, dass eine optimale Lösung die folgende Form hat:

$$(x_1, \ldots, x_n) := (\underbrace{1, 1, \ldots, 1}_{k}, b, \underbrace{0, 0, \ldots, 0}_{n-k-1})$$

  Dabei hat b den Wert $(M - \sum_{i=1}^{k} w_i)/w_{k+1}$ und die Lösung erreicht den Wert $\sum_{i=1}^{k} p_i + b \cdot p_{i+1}$.
- Reduzieren wir eine der ersten k Einsen, dann können wir nur mit einem Objekt $j > k$ den Rucksack auffüllen, also x_j erhöhen.
- Die neue Lösung wird um $(1 - \alpha) \cdot p_i$ reduziert und um $\beta \cdot p_j$ erhöht, wobei $\beta \cdot w_j = (1 - \alpha) \cdot w_i$, also $\beta = (1 - \alpha) \cdot w_i / w_j$ gilt.

Fortsetzung:

- Wir hatten auf der letzten Seite festgestellt: Die neue Lösung wird um $(1-\alpha)\cdot p_i$ reduziert und um $\beta \cdot p_j$ erhöht, wobei $\beta \cdot w_j = (1-\alpha)\cdot w_i$, also $\beta = (1-\alpha)\cdot w_i/w_j$ gilt.
- Der Wert der neuen Lösung unterscheidet sich also vom Wert der alten Lösung um $(1-\alpha)\cdot w_i p_j/w_j - (1-\alpha)\cdot p_i$.
- Dieser Betrag ist kleiner gleich Null, da $p_i/w_i \geq p_j/w_j$ gilt.

$\rightarrow$ Die neue Lösung ergibt keine Verbesserung.

analog zeigt man: Man erhält keine Verbesserung, wenn der Wert $x_{k+1} = b$ reduziert und stattdessen ein Wert x_j mit $j > k+1$ erhöht wird.

Coin Changing

The problem is to make change for n cents using the least number of coins. Is also called Change Making Problem CMP.

Example: Consider $n = 88c$ and coins $25c$, $10c$, $5c$, $1c$.

i	c_i	n_i	n_i div c_i	$n_{i+1} = n_i \bmod c_i$
1	25	88	3	13
2	10	13	1	3
3	5	3	0	3
4	1	3	3	0

$\rightarrow 88c = 3 \cdot 25c + 1 \cdot 10c + 0 \cdot 5c + 3 \cdot 1c$

Counter-Example: Consider $n = 14c$ and coins $11c$, $7c$, $1c$.

- greedy yields $n = 1 \cdot 11c + 3 \cdot 1c \rightarrow 4$ coins
- optimal is $n = 2 \cdot 7c \rightarrow 2$ coins

Counter-Example: Consider $n = 34c$ and coins $25c$, $10c$, $1c$.

- greedy yields $n = 1 \cdot 25c + 9 \cdot 1c \rightarrow 10$ coins
- optimal is $n = 3 \cdot 10c + 4 \cdot 1c \rightarrow 7$ coins

Frage: Gilt das Optimalitätsprinzip von Bellman?

Übung 22.

- *Show that the greedy algorithm always yields an optimal solution for the coins* $25c$, $10c$, $5c$, $1c$.
- *Suppose that the available coins are in the denominations* $c^0, c^1, c^2, \ldots, c^k$ *for some integers* $c > 1$ *and* $k \geq 1$. *Show that the greedy algorithm always yields an optimal solution.*

Antwort: Das Optimalitätsprinzip von Bellman gilt. Beispiel:

- $(20, 10, 5, 2, 1)$ ist optimal für das deutsche Münzsystem und den Betrag 38 Cent.
- Dann muss $(10, 5, 2, 1)$ optimal sein für $38 - 20$ Cent, denn gäbe es eine Lösung (b_1, b_2, b_3) mit weniger Münzen und $b_1 + b_2 + b_3 = 18$ Cent, dann wäre $(20, b_1, b_2, b_3)$ eine bessere Lösung für 38 Cent. $\lightning$

Wir führen also einen Beweis durch Widerspruch:

- Sei $(a_1, \ldots, a_m)$ mit $a_i \in \{d_1, \ldots, d_k\}$ eine optimale Lösung für den Wert n, wobei $\sum_{i=1}^{m} a_i = n$ gilt, also alle Münzen zusammen den Wert n ergeben, und die Münzen aus dem gegebenen Wertebereich sind.
- Wenn wir die Münze a_1 wegnehmen, dann ist $(a_2, \ldots, a_m)$ eine optimale Lösung für den Wert $n - a_1$.

  Andernfalls gäbe es eine Lösung $(b_1, \ldots, b_k)$ mit $k < m - 1$ für den Wert $n - a_1$. Dann wäre aber $(a_1, b_1, b_2, \ldots, b_k)$ eine bessere Lösung für den Wert n. $\lightning\lightning$

Coin Changing

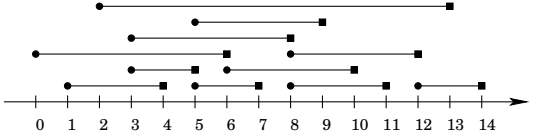
Remarks:

- Change Making Problem is NP-complete! It is a variant of the knapsack problem, where each item can be taken multiple times. [Lueker, 1975]
- Coin systems, for which the greedy algorithm always gives an optimal solution, are called canonical.
- Goebbels, Gurski, Rethmann, Yilmaz. Change-Making Problems revisited: A Parameterized Point of View. Journal of Combinatorial Optimization, 34(4): 1218-1236, 2017.

Gegeben: Eine Menge $S = \{a_1, \ldots, a_n\}$ von n Aktivitäten, die alle die gleiche Ressource benötigen. Eine Aktivität a_i hat stets einen Beginn $b(a_i)$ und ein Ende $e(a_i)$.

Gesucht: Eine möglichst große Menge paarweise kompatibler Aktivitäten. Zwei Aktivitäten a_i und a_j heißen kompatibel, wenn sich die Zeitintervalle nicht überschneiden, also $[b(a_i), e(a_i)[\cap [b(a_j), e(a_j)[= \emptyset$ gilt.

Annahme: Die Aktivitäten sind anhand der Endzeitpunkte aufsteigend sortiert, also $e(a_1) \leq e(a_2) \leq \ldots \leq e(a_n)$.



Übung 23. *Formulieren Sie einen Algorithmus nach folgender Idee: Wähle stets diejenige Aktivität mit frühestem Endzeitpunkt, die legal eingeplant werden kann!*

Welche Laufzeit hat Ihr Algorithmus? Liefert das Greedy-Verfahren eine optimale Lösung?

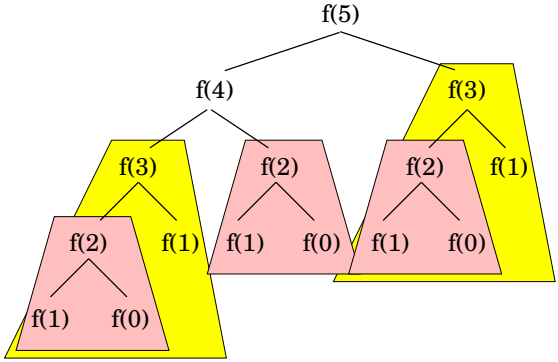
Übersicht

Motivation: Berechne Fibonacci-Zahlen rekursiv, also top-down.

```
long fibo(int n) {
   long f1, f2;

   if (n <= 1)
       return n;

   f1 = fibo(n-1);
   f2 = fibo(n-2);
   return f1 + f2;
}
```



Problem: Viele Zwischenlösungen werden mehrfach berechnet! Dadurch ist die Laufzeit sehr groß.

Achtung: Rekursion ist immer dann zu vermeiden, wenn sich die rekursiv zu lösenden Teilprobleme überlappen.

Lösung: Speichere bereits berechnete Zwischenlösungen in einer Tabelle. Diese Technik nennt man *Memorieren*. Es ist immer noch ein Top-Down-Ansatz.

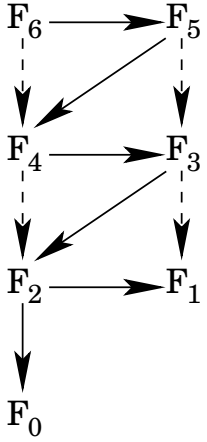
```
long fibo(int n) {
   long f1, f2;

   if (fib[n-1] >= 0)
      f1 = fib[n-1];
   else {
      f1 = fibo(n-1);
      fib[n-1] = f1;
   }
   if (fib[n-2] >= 0)
      f2 = fib[n-2];
   else {
      f2 = fibo(n-2);
      fib[n-2] = f2;
   }

   return f1 + f2;
}
```

Frage: Wie kann das Programm vereinfacht werden?

Die zweite Fallunterscheidung kann entfallen: Nachdem `fib(n-1)` berechnet wurde, ist in `fib[n-2]` der korrekte Wert gespeichert. Dadurch ergibt sich folgende Struktur der rekursiven Aufrufe:



Gestrichelte Linien stellen den Zugriff auf bereits gespeicherte, memorierte Werte dar.

Bottom-Up-Ansatz: Aus Rekursion wird Iteration, indem aus kleinen Teillösungen größere Lösungen berechnet werden.

Die Berechnung beginnt beim Rekursionsende, da nur diese Werte initial bekannt sind. Während der Berechnung muss sichergestellt werden, dass die in diesem Schritt benötigten Teillösungen bereits berechnet wurden.

```
long fibo(unsigned int n) {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i <= n; i++)
        fib[i] = fib[i-1] + fib[i-2];

    return fib[n];
}
```

```
fib[2] = fib[1] + fib[0]
fib[3] = fib[2] + fib[1]
fib[4] = fib[3] + fib[2]
fib[5] = fib[4] + fib[3]
fib[6] = fib[5] + fib[4]
fib[7] = fib[6] + fib[5]
...
```

Allerdings ist dies kein Beispiel für dynamische Programmierung: Es fehlt die Optimierung!

Im Gegensatz zur Greedy-Methode speichert die Methode der dynamischen Programmierung alle bereits berechneten Teillösungen in einer Tabelle (Bottom-Up).

- Aus den Teillösungen wird die Gesamtlösung zusammengesetzt, indem alle Kombinationen von Teillösungen betrachtet werden und die optimale Lösung ausgewählt wird.

$\rightarrow$ Die Laufzeit ist in der Regel höher als bei einem Greedy-Ansatz.

Auch bei der dynamischen Programmierung wird nur dann eine optimale Lösung berechnet, wenn das *Optimalitätsprinzip nach Bellman* gilt: „Die optimale Lösung eines Problems setzt sich aus optimalen Lösungen kleinerer Teilprobleme zusammen.“

Da alle Kombinationen getestet werden und das Optimalitätsprinzip nach Bellman gilt, muss die Optimalität der Lösung nicht gezeigt werden. Es muss allerdings gezeigt werden, dass das Optimalitätsprinzip nach Bellman gilt, was aber oft einfach ist, siehe bspw. Change-Making-Problem.

Beispiel: Rekursiver Ansatz für das Wechselgeldproblem, wenn es mit Greedy nicht optimal zu lösen ist. Gegeben seien die Münzen $d_1 = 11$, $d_2 = 5$ und $d_3 = 1$ sowie der Betrag $p = 15$.

In einem Schritt werden alle möglichen Münzen gewählt und die optimale Darstellung des Restbetrags (rekursiv) berechnet:

- Wähle d_1 $\rightarrow$ Restbetrag $15 - 11 = 4$ benötigt 4 Münzen: $4 = 4 \cdot 1$
  - $\rightarrow$ Kombination benötigt 5 Münzen, einmal d_1 sowie die Münzen für den Restbetrag.
- Wähle d_2 $\rightarrow$ Restbetrag $15 - 5 = 10$ benötigt 2 Münzen: $10 = 2 \cdot 5$
  - $\rightarrow$ Kombination benötigt 3 Münzen, einmal d_2 sowie die Münzen für den Restbetrag.
- Wähle d_3 $\rightarrow$ Restbetrag $15 - 1 = 14$ benötigt 4 Münzen: $14 = 1 \cdot 11 + 3 \cdot 1$
  - $\rightarrow$ Bei dieser Kombination werden 5 Münzen benötigt.

In jedem Schritt (auch bei der Rekursion) wird immer der minimale Wert gewählt.

Betrachten wir ein Münzsystem mit den Werten $d_1, \ldots, d_k$. Sei $C(p)$ die minimale Anzahl an Münzen, um den Betrag p auszuzahlen. Rekursiver Ansatz:

$$C(p) = \begin{cases} 0 & \text{if } p = 0 \\ \min\limits_{i:d_i \leq p} \{1 + C(p - d_i)\} & \text{if } p > 0 \end{cases}$$

Beim Buttom-Up-Ansatz für das Wechselgeldproblem beginnt die Berechnung beim Rekursionsende, also $p = 0$:

$C[0] := 0$
for $p := 1$ to n **do**
    $min := \infty$
    for $i := 1$ to k **do**
        if $p - d[i] \geq 0$ **and** $1 + C[p - d[i]] < min$ **then**
            $min := 1 + C[p - d[i]]$
    $C[p] := min$

Matrix-Kettenmultiplikation

Gegeben: n Matrizen $M_1, \ldots, M_n$, wobei M_i eine $p_{i-1} \times p_i$ Matrix ist, also p_{i-1} Zeilen und p_i Spalten hat.

Gesucht: Eine Klammerung, so dass $M_1 \cdot \ldots \cdot M_n$ mit minimaler Anzahl an skalaren Multiplikationen erfolgt.

Anmerkung: Multiplikation $p \times q$ Matrix mit $q \times r$ Matrix erfordert $p \cdot q \cdot r$ Multiplikationen und liefert $p \times r$ Matrix.

Beispiel: M_1, M_2, M_3 haben die Dimensionen (50×10), (10×20) und (20×5). Dann sind bei

- $(M_1 \cdot M_2) \cdot M_3 \;\rightarrow\; 50 \cdot 10 \cdot 20 + 50 \cdot 20 \cdot 5 = 15.000$
- $M_1 \cdot (M_2 \cdot M_3) \;\rightarrow\; 10 \cdot 20 \cdot 5 + 50 \cdot 10 \cdot 5 = 3.500$

Multiplikationen erforderlich!

Naiv: Ausprobieren aller möglichen Klammerungen.

$$K_n = \sum_{j=1}^{n-1} K_j \cdot K_{n-j} \quad \text{mit} \quad K_1 = 1$$

Erklärung:

- Es gibt $n-1$ mögliche äußere Klammerungen.
- Bei Aufteilung in $M_1 \cdots M_j$ und $M_{j+1} \cdots M_n$ verbleiben K_j bzw. K_{n-j} innere Klammerungen.

Lösung der Rekursionsgleichung: Catalansche Zahlen

$$K_{n+1} = C(n) = \binom{2n}{n} \frac{1}{n+1} \in \Omega\left(\frac{4^n}{n^{1.5}}\right)$$

Bezeichne $m(i,j)$ die minimale Anzahl skalarer Multiplikationen bei optimaler Klammerung des Abschnitts $M_i \cdots M_j$.

Sei $(M_i \cdots M_k) \cdot (M_{k+1} \cdots M_j)$ die optimale Klammerung des Abschnitts $M_i \cdots M_j$. Dann gilt:

$$m(i,j) = m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j$$

→ *Optimalitätsprinzip:* Eine optimale Lösung kann aus optimalen Teillösungen zusammengesetzt werden.

Da wir die optimale Klammerung nicht kennen, müssen wir alle möglichen Werte für k ausprobieren und dann den minimalen Wert wählen. Damit ergibt sich folgende Rekursionsformel:

$$m(i,j) = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j} (m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j), & i < j \end{cases}$$

Übung 24.

1. *Formulieren Sie einen rekursiven Algorithmus zur Lösung der Matrix-Kettenmultiplikation. Welche Laufzeit hat Ihr Algorithmus?*
2. *Wie sieht eine Lösung mittels dynamischer Programmierung aus und welche Laufzeit hat diese Lösung?*
3. *Wie kann der Algorithmus erweitert werden, so dass auch die optimale Klammerung bestimmt wird?*

0/1-Rucksack-Problem

Packe einen Teil von n Objekten mit den Größen $g_1, \ldots, g_n$ und den Werten $w_1, \ldots, w_n$ so in einen Rucksack der Größe G, dass der Gesamtwert maximal ist.

Formal: Finde einen 0/1-Vektor $(a_1, \ldots, a_n) \in \{0,1\}^n$ mit

$$\sum_{i=1}^{n} a_i g_i \leq G \quad \text{und} \quad \sum_{i=1}^{n} a_i w_i \longrightarrow \max.$$

Formulierung als Integer Programm!

Rekursiver Algorithmus: Sei $knap(h, i)$ der maximale Wert, der mit den Objekten $i, \ldots, n$ und Rucksackgröße h erreicht werden kann. Dann gilt für $i < n$ und $h \geq g_i$:

$$knap(h, i) = \max\{knap(h, i+1),\ w_i + knap(h - g_i, i+1)\}$$

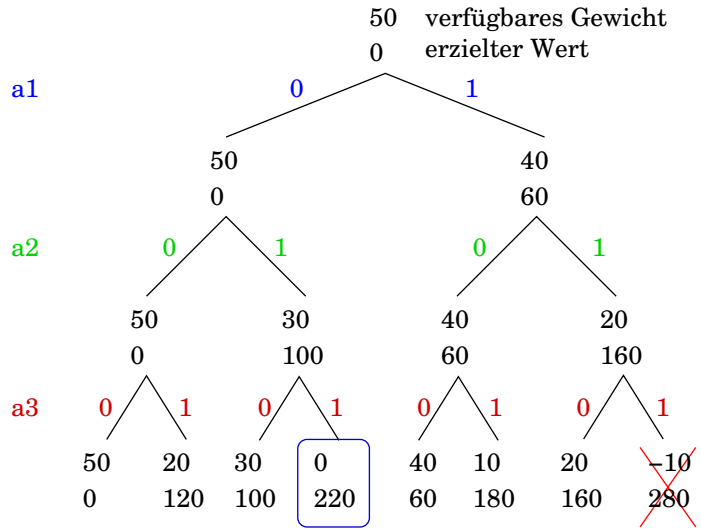
Sonst gilt:

$$knap(h, i) = \begin{cases} knap(h, i+1), & i < n, h < g_i \\ 0, & i = n, h < g_n \\ w_n, & i = n, h \geq g_n \end{cases}$$

Übung 25.

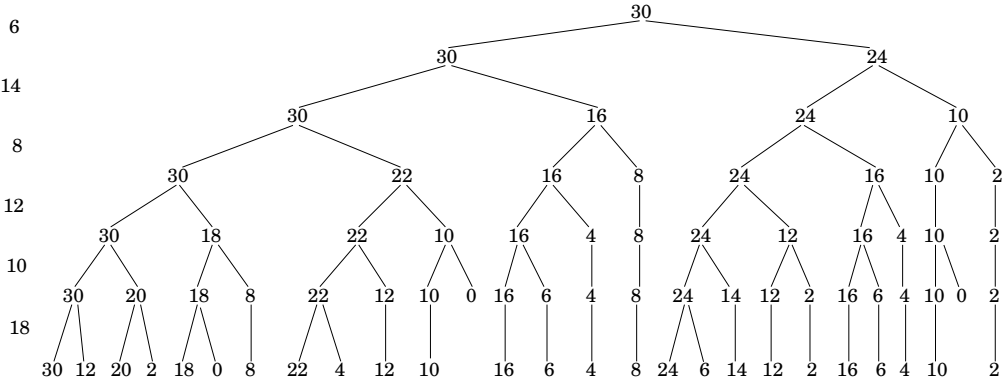
1. *Implementieren Sie obigen Algorithmus. Laufzeit?*
2. *Lösung mittels dynamischer Programmierung? Laufzeit?*

Beispiel: $g = (10, 20, 30)$, $w = (60, 100, 120)$, $G = 50$



Probleme:

1. Laufzeit des rekursiven Algorithmus: $\mathcal{O}(2^n)$
2. Memorieren bringt nichts, da nur wenige Teillösungen wiederverwendet werden können. Beispiel: rekursive Aufrufe zu $g = (6, 14, 8, 12, 10, 18)$ mit $G = 30$.



Versuch: Bottom-Up-Strategie

```
FOR i := n DOWN TO 1 DO
    FOR h := 0 TO G DO
        berechne knap(h, i) nach obiger Formel
```

$\longrightarrow$ Laufzeit $\in \mathcal{O}(n \cdot G)$

Beispiel: $g = (1, 2, 3)$, $w = (6, 10, 12)$, $G = 5$

$i \backslash h$	0	1	2	3	4	5
3	0	0	0	12	12	12
2	0	0	10	12	12	22
1	0	6	10	16	18	22

Übung 26.

- *Welcher Algorithmus ist besser? Der rekursive mit Laufzeit $\mathcal{O}(2^n)$ oder der mittels dynamischer Programmierung und Laufzeit $\mathcal{O}(n \cdot G)$?*
- *Ist die Laufzeit von $\mathcal{O}(n \cdot G)$ nicht ein Widerspruch zur Tatsache, dass das 0/1-Rucksack-Problem NP-vollständig ist?*

Optimale Suchbäume

Zur Speicherung von Werten wurden im Bachelorstudium balancierte Suchbäume wie AVL- oder Rot-Schwarz-Bäume vorgestellt. Mittels sogenannter Rotationen wird eine logarithmische Tiefe der Bäume sichergestellt, sodass Operationen wie Einfügen, Suchen und Entfernen effizient möglich sind.

Splay-Bäume (siehe Bachelorstudium) sind selbstanordnende Suchbäume: Jeder Schlüssel, auf den zugegriffen wurde, wird mittels Umstrukturierungen zur Wurzel bewegt. Dadurch erfolgt eine Anpassung an unterschiedliche Zugriffshäufigkeiten, wobei die Zugriffshäufigkeiten vorher nicht bekannt sind.

- Oft angefragte Schlüssel werden in Richtung Wurzel bewegt.
- Selten angefragte Schlüssel wandern zu den Blättern hinab.

Wenn die Zugriffswahrscheinlichkeiten oder -häufigkeiten bekannt sind, können wir optimale Suchbäume in dem Sinne generieren, dass die Kosten zum Suchen der Elemente minimal sind.

gegeben:

- Menge $S = \{k_1, \ldots, k_n\} \subseteq U$ von n verschiedenen Schlüsseln $k_1 < k_2 < \ldots < k_n$ aus einem endlichen, geordneten Universum $U = (k_0, k_{n+1})$, wobei aber auch $k_0 = -\infty$ und $k_{n+1} = +\infty$ möglich ist.
- (absolute) Häufigkeit a_i, mit der nach $k_i \in S$ gesucht wird
- Für erfolglose Suchanfragen $k \notin S$ mit $k_i < k < k_{i+1}$ fügen wir zusätzlich Dummy-Knoten d_i ein (eigentlich sind das Blätter).
- (absolute) Häufigkeit b_i, mit der nach $k \notin S$ mit $k_i < k < k_{i+1}$ gesucht wird

Das Gewicht eines Baums T ist definiert als:

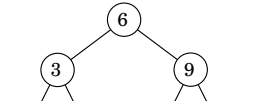
$$W(T) = \sum_{i=1}^{n} a_i + \sum_{j=0}^{n} b_j$$

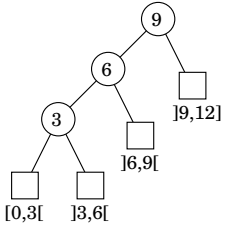
Gesucht: Binärer Suchbaum, der die Suchkosten (die gewichtete Pfadlänge) minimiert.

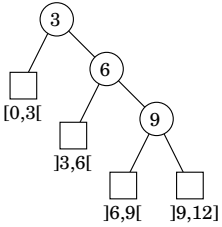
$$P(T) = \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot a_i + \sum_{j=0}^{n} \text{depth}_T(d_j) \cdot b_j \longrightarrow \min$$

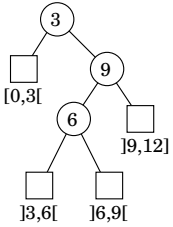
Wir gehen davon aus, dass die Wurzel eine Tiefe von Eins hat.

Beispiel: $S = \{3, 6, 9\}$ aus dem Intervall $(0, 12)$, wobei $a_1 = 2$, $a_2 = 4$, $a_3 = 14$ die Zugriffshäufigkeiten der Schlüssel und $b_0 = b_1 = b_2 = b_3 = 1$ die Zugriffshäufigkeiten auf die nicht vorhandenen Schlüssel sind.









$$P(T_1) = 1 \cdot a_2 + 2 \cdot (a_1 + a_3) + 3 \cdot (b_0 + b_1 + b_2 + b_3) = 48$$

$$P(T_2) = 1 \cdot a_3 + 2 \cdot (a_2 + b_3) + 3 \cdot (a_1 + b_2) + 4 \cdot (b_0 + b_1) = 41$$

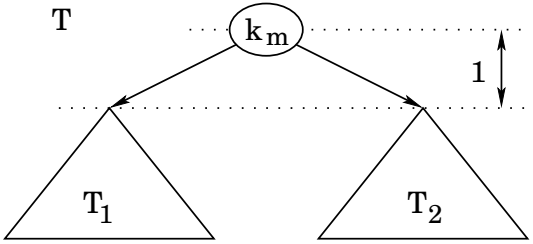
$$P(T_3) = 1 \cdot a_1 + 2 \cdot (b_0 + a_2) + 3 \cdot (b_1 + a_3) + 4 \cdot (b_2 + b_3) = 65$$

$$P(T_4) = 1 \cdot a_1 + 2 \cdot (b_0 + a_3) + 3 \cdot (a_2 + b_3) + 4 \cdot (b_1 + b_2) = 55$$

Sei T ein binärer Suchbaum mit Wurzel k_m und den Teilbäumen T_1 und T_2. Dann gilt:

$$\begin{aligned} P(T) &= P(T_1) + W(T_1) + a_m + P(T_2) + W(T_2) \\ &= P(T_1) + P(T_2) + W(T) \end{aligned}$$

denn für einen Knoten $x \in T_1$ gilt $\text{depth}_T(x) = \text{depth}_{T_1}(x) + 1$ und für $x \in T_2$ gilt $\text{depth}_T(x) = \text{depth}_{T_2}(x) + 1$.



Es gilt also das Optimalitätsprinzip von Bellman: Jeder Teilbaum eines optimalen Suchbaums ist selbst ein optimaler Suchbaum.

Bezeichnungen:

$T(i,j)$ optimaler Suchbaum für $\{d_i, k_{i+1}, \ldots, k_j, d_j\}$
$W(i,j)$ das Gewicht von $T(i,j)$, also $W(i,j) = b_i + a_{i+1} + \ldots + a_j + b_j$
$P(i,j)$ die gewichtete Pfadlänge von $T(i,j)$

Diese Werte sind definiert für alle $0 \leq i \leq j$. Für $j = i$ besteht $T(i,i)$ nur aus dem Blatt d_i mit Häufigkeit b_i und Tiefe 1. Wir erhalten folgende Gleichungen:

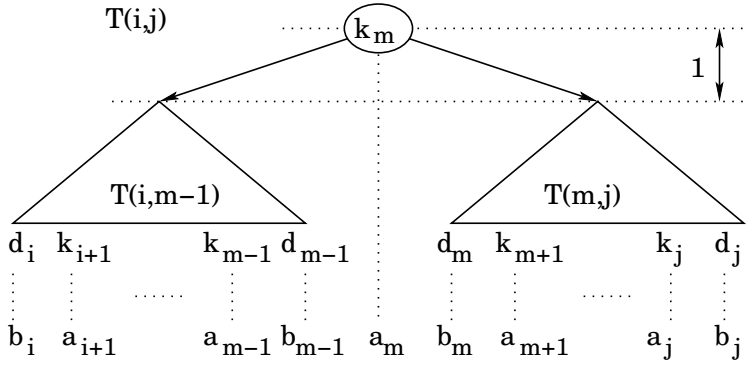
$$W(i,j) := \begin{cases} b_i & \text{falls } i = j \\ W(i,j-1) + a_j + b_j & \text{falls } i < j \end{cases}$$

und

$$P(i,j) := \begin{cases} b_i & \text{falls } i = j \\ W(i,j) + \min\limits_{i<m\leq j} \{P(i,m-1) + P(m,j)\} & \text{falls } i < j \end{cases}$$

Da wir nicht wissen, welcher Knoten k_m die Wurzel des Baums $T(i,j)$ ist, müssen alle Möglichkeiten berechnet werden.

noch einmal die letzte Rekursion: $P(i,j) := W(i,j) + \min_{i<m\leq j} \{P(i,m-1) + P(m,j)\}$



Auch diese Rekursion können wir iterativ bottom-up berechnen, indem wir die Differenz $d := j - i$ von 0 ausgehend vergrößern und die rekursiven Aufrufe durch Array-Zugriffe ersetzen.

```
1:  for i := 0 to n do
2:      W[i][i] := b_i
3:      P[i][i] := b_i
4:  for d := 1 to n do
5:      for i = 0 to n − d do
6:          j = i + d
7:          W[i][j] = W[i][j − 1] + a_j + b_j
8:          P[i][j] = ∞
9:          for m := i to j − 1 do
10:             q := W[i][j] + P[i][m − 1] + P[m][j]
11:             if (q < P[i][j]) then
12:                 P[i][j] := q
13:                 r[i][j] := m
```

Die Zahl $r[i][j]$ speichert den Index der Wurzel von $T(i,j)$, also den Index, für das das Minimum angenommen wird. Gesucht ist $T(0,n)$.

Laufzeit:

$$
\begin{aligned}
T(n) &\leq \sum_{d=1}^{n}\sum_{i=0}^{n-d}\left(c_1 + \sum_{m=i}^{i+d-1} c_2\right) = \sum_{d=1}^{n}\sum_{i=0}^{n-d}\left(c_1 + d\cdot c_2\right) \\
&\leq \sum_{d=1}^{n}\sum_{i=0}^{n-d} c\cdot d = c\cdot\sum_{d=1}^{n}\sum_{i=0}^{n-d} d = c\cdot\sum_{d=1}^{n}(n-d+1)\cdot d \\
&= \mathcal{O}\left(\sum_{d=1}^{n}((n+1)d - d^2)\right) = \mathcal{O}\left((n+1)\sum_{d=1}^{n} d - \sum_{d=1}^{n} d^2\right) = \mathcal{O}(n^3)
\end{aligned}
$$

denn es gilt

$$
\sum_{d=1}^{n} d = \frac{n\cdot(n+1)}{2} \in \mathcal{O}(n^2) \quad \text{und} \quad \sum_{d=1}^{n} d^2 = \frac{n\cdot(n+1)\cdot(2n+1)}{6} \in \mathcal{O}(n^3).
$$

Insgesamt ergibt sich als Laufzeit $\mathcal{O}(n^3)$.

Eine Verbesserung der Laufzeit erhält man, wenn das Monotonieprinzip[7] erfüllt ist.

Ohne Beweis:

$$r[i][j-1] \;\leq\; r[i][j] \;\leq\; r[i+1][j]$$

Dann genügt es, in Zeile 9 die for-Schleife für den Bereich $r[i][j-1] \leq m \leq r[i+1][j]$ zu betrachten.

Auf der nächsten Seite wird gezeigt, dass sich damit insgesamt nur eine quadratische Laufzeit ergibt: $\mathcal{O}(n^2)$

[7] D.E. Knuth. The Art of Computer Programming, Vol.3: Sorting and Searching. Addison-Wesley, 1973.

Für festes d ergibt sich $\mathcal{O}(n)$ als Laufzeit der for-Schleife in Zeile 5:

$$
\begin{aligned}
T(n) &\leq \sum_{i=0}^{n-d} (c_1 + c_2 \cdot (r(i+1, i+d) - r(i, i+d-1) + 1)) \\
&= c_1 \cdot (n-d+1) + c_2 \cdot \sum_{i=0}^{n-d} (r(i+1, i+d) - r(i, i+d-1) + 1)) \\
&= \mathcal{O}(n) + c_2 \cdot \big(r(1,d) - r(0,d-1) + 1 \;+\; r(2,d+1) - r(1,d) + 1 \\
&\quad + r(3,d+2) - r(2,d+1) + 1 \;+\; r(4,d+3) - r(3,d+2) + 1 \\
&\quad \ldots + \; r(n-d+1, n) - r(n-d, n-1) + 1 \big) \\
&= \mathcal{O}(n) + \underbrace{c_2 \cdot (n-d+1)}_{\mathcal{O}(n)} + c_2 \cdot (\underbrace{r(n-d+1, n) - r(0, d-1)}_{\leq n}))
\end{aligned}
$$

Insgesamt erhalten wir als Laufzeit $\mathcal{O}(n^2)$.

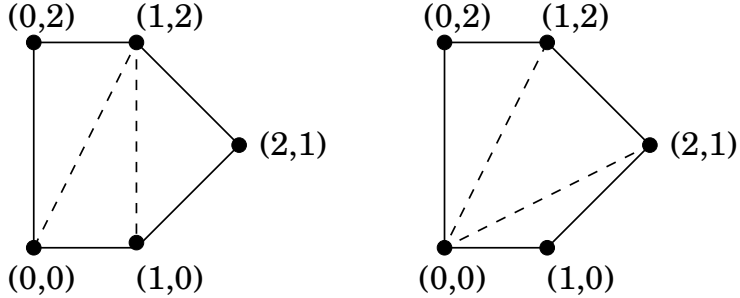
Das Monotonieprinzip lässt sich auch bei der Matrix-Kettenmultiplikation anwenden.

Gegeben: Ein konvexes Polygon aus Punkten $(p_1, \ldots, p_n)$.

Gesucht: Eine Teilmenge B der Menge aller Saiten (auch Sehnen genannt) mit den Eigenschaften:

- Keine zwei Saiten aus B kreuzen sich und
- die Saiten aus B teilen das Polygon in Dreiecke.

Beispiel:



Sei $w(i,j,k)$ der Umfang des Dreiecks $\triangle p_i p_j p_k$. Die Kosten einer Triangulierung ist die Summe über den Umfang aller Dreiecke.

Im Beispiel:

$$\begin{array}{rl} & (1+2+\sqrt{5}) \\ + & (1+2+\sqrt{5}) \\ + & (2+\sqrt{2}+\sqrt{2}) \\ \hline = & 8+2\sqrt{2}+2\sqrt{5} \\ \approx & 15.30 \end{array} \qquad \begin{array}{rl} & (1+2+\sqrt{5}) \\ + & (\sqrt{2}+\sqrt{5}+\sqrt{5}) \\ + & (1+\sqrt{2}+\sqrt{5}) \\ \hline = & 4+2\sqrt{2}+4\sqrt{5} \\ \approx & 15.77 \end{array}$$

Übung 27.

- *Formulieren Sie einen rekursiven Algorithmus für das Problem. Welche Laufzeit hat Ihr Algorithmus?*
- *Lösung mittels dynamischer Programmierung? Laufzeit?*

Levenshtein-Distanz

Die Levenshtein-Distanz zweier Wörter a und b wird auch als Edit-Distanz, Editierdistanz oder Editierabstand bezeichnet.

Sie bezeichnet ein Maß für den Unterschied zwischen zwei Zeichenketten bezüglich der minimalen Anzahl der Operationen Einfügen, Löschen und Ersetzen, um die eine Zeichenkette in die andere zu überführen.

Beispiel: Die Edit-Distanz von `Tier` und `Tor` ist zwei.

$$\texttt{Tier} \rightarrow \texttt{Toer} \text{ (ersetze i durch o)} \rightarrow \texttt{Tor} \text{ (Lösche e)}$$

In der Praxis: Bestimmen der Ähnlichkeit von Zeichenketten zur Rechtschreibprüfung oder bei einer Duplikaterkennung.

Rekursiver Algorithmus: Sei $D_{i,j}$ der minimale Editierabstand zwischen den Teilwörtern bis Position i bzw. j. Dann gilt initial:

$$D_{0,0} = 0, \qquad D_{0,j} = j, \qquad D_{i,0} = i$$

Außerdem gilt:

$$D_{i,j} = \min \left\{ \begin{array}{cc} D_{i,j-1} + 1 & \text{einfügen} \\ D_{i-1,j} + 1 & \text{löschen} \\ D_{i-1,j-1} + c & \text{sonst} \end{array} \right\}$$

Dabei ist $c = 0$, falls $a_i = b_j$ gilt, sonst ist $c = 1$ (ersetze a_i durch b_j).

Übung 28.

1. *Gilt das Optimalitätsprinzip?*
2. *Implementieren Sie obigen rekursiven Algorithmus und eine Lösung mittels dynamischer Programmierung.*
3. *Vergleichen Sie die beiden Laufzeiten für einige Beispiele.*

Längste gemeinsame Teilfolge

Eine Teilfolge kann durch Streichen beliebiger, nicht notwendig benachbarter Zeichen erzeugt werden:

Gegeben: Zwei Zeichenfolgen $A = (a_1, \ldots, a_m)$ und $B = (b_1, \ldots, b_n)$

Gesucht: Längste gemeinsame Teilfolge von A und B.

Beispiel: Die längste gemeinsame Teilfolge von ANANAS und BANANENMUS ist ANANS und hat die Länge 5: ANANAS und BANANENMUS

Naiv: Erzeuge alle möglichen Teilfolgen von A und teste für jede solche Teilfolge Z, ob Z auch eine Teilfolge von B ist. Vermerke zu jedem Schritt die bisher längste gemeinsame Teilfolge. Laufzeit in $\Omega(2^m)$.

Übung 29. *Gilt das Optimalitätsprinzip von Bellman? Geben Sie eine Rekursionsformel zur Lösung des Problems an. Formulieren Sie dann einen iterativen Algorithmus.*

Traveling Salesperson Problem

Bestimme zu einem vollständigen Graphen $G = (V, E, c)$ einen Hamiltonkreis, dessen Gesamtlänge möglichst klein ist. Dabei ist $c : E \to \mathbb{N}$ eine Kostenfunktion, die zu jeder Kante in E deren „Länge“ definiert.

Da der Knoten, in der die Rundreise beginnt, festgelegt ist, hat die erschöpfende Suche bei n Knoten eine Komplexität von $(n-1)!$.

$$(1,2,3,4,1) \quad (1,3,2,4,1) \quad (1,4,3,2,1)$$
$$(1,2,4,3,1) \quad (1,3,4,2,1) \quad (1,4,2,3,1)$$

Geht es effizienter?

Gilt das Optimalitätsprinzip von Bellman?

aus dem Buch *Algorithmik* von Uwe Schöning:

- Wenn eine optimale Rundreise bei Knoten 1 beginnt und dann den Knoten k besucht, dann muss der Weg von k aus durch die Städte $\{2, \ldots, n\} - \{k\}$ zurück nach Knoten 1 auch optimal sein.
- Sei $g(i, S)$ die Länge des kürzesten Wegs, der bei Knoten i beginnt, dann durch jeden Knoten aus S genau einmal führt, und dann bei Knoten 1 endet.
- Berechne $g(1, \{2, \ldots, n\})$ mittels dynamischer Programmierung und folgender Formel:

$$g(i, S) = \begin{cases} c(i, 1) & \text{falls } S = \emptyset \\ \min\limits_{j \in S} \Big(c(i, j) + g(j, S - \{j\}) \Big) & \text{sonst} \end{cases}$$

  Da wir nicht wissen, welcher Knoten j dem Knoten i auf der optimalen Rundreise folgt, werden alle möglichen Knoten getestet.
- Laufzeit: $\mathcal{O}(n^2 \cdot 2^n)$ $\leftarrow$ Tabellengröße mal Aufwand pro Eintrag

Fazit:

- Bestimme die rekursive Struktur einer optimalen Lösung.
- Entwerfe eine rekursive Methode zur Bestimmung des Werts einer optimalen Lösung.
- Transformiere die rekursive Methode in eine iterative Methode zur Bestimmung des Werts einer optimalen Lösung.
  $\rightarrow$ Vermeide mehrfache Berechnung von Teilergebnissen.
- Bestimme aus dem Wert einer optimalen Lösung und den im vorigen Schritt berechneten Zusatzinformationen eine optimale Lösung.

Übersicht

Lokale Suche

Idee:

- Beginne mit einer beliebigen Lösung.
- Mutiere die Lösung durch geringfügige, lokale Veränderungen.
- Ist die neue Lösung besser, übernehme die neue Lösung.
- Das Verfahren wird auch *hill climbing* oder *lokale Verbesserungsstrategie* genannt.

Problem: Eventuell wird nur ein lokales Optimum gefunden.

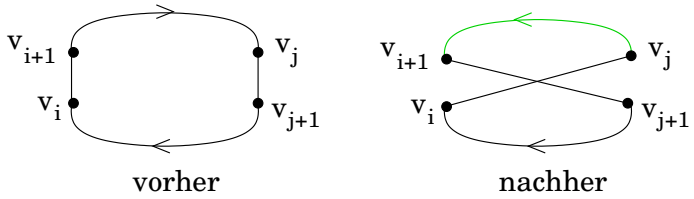
Lösung:

- Wiederholen mit unterschiedlichen Startlösungen.
- Wähle lokale Veränderungen zufällig aus.
- Akzeptiere mit geringer Wahrscheinlichkeit auch schlechtere Lösungen.

- Initial: Wähle zufällige Permutation der Knoten $v_1, v_2, \ldots, v_n$.
- Wähle zwei zufällige Knoten v_i und v_j, die auf der bisherigen Rundreise nicht aufeinander folgen, dabei sei $i < j$.
- Falls die neue Rundreise

$$v_1, v_2, \ldots, v_i, v_j, v_{j-1}, \ldots, v_{i+2}, v_{i+1}, v_{j+1}, v_{j+2}, \ldots, v_n$$

kürzer ist, wiederhole das Verfahren mit der geänderten Rundreise.



- Nur bei symmetrischer Entfernungsmatrix sinnvoll.

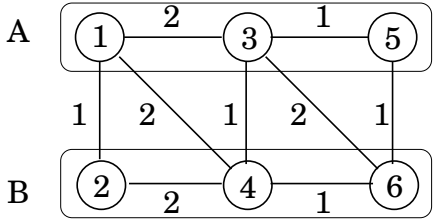
Übung 30.

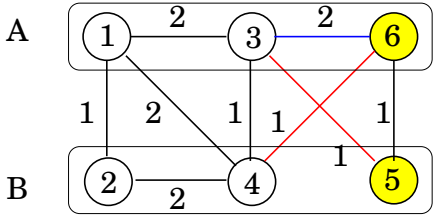
- *Schreiben Sie eine Funktion, die eine zufällige Permutation von n Zahlen berechnet.*
- *Schreiben Sie eine Funktion, die alle Permutationen von n Zahlen aufzählt.*

- Given a symmetric cost matrix $[d_{ij}]$ defined on the edges of an undirected graph $G = (V, E)$ with $|V| = 2n$.
- A partition $V = A \uplus B$ such that $|A| = |B| = n$ is called a uniform partition.
- Find a uniform partition such that the cost

$$c(A, B) = \sum_{i \in A, j \in B} d_{ij}$$

is minimum over all uniform partitions.





Uniform Graph Partitioning is NP-complete[8].

Observation: Given a uniform partition A, B.

- Suppose A^*, B^* is an optimal uniform partition.
- Let X be those elements of A that are not in A^* and let Y be similarly defined for B.
- Then $|X| = |Y|$ and $A^* = (A \backslash X) \cup Y$ and $B^* = (B \backslash Y) \cup X$.

For two elements $a \in A, b \in B$, the operating of forming $A' := (A \backslash \{a\}) \cup \{b\}$ and $B' := (B \backslash \{b\}) \cup \{a\}$ is called a swap.

[8] M.R. Garey, D.S. Johnson, L. Stockmeyer. Some Simplified NP-complete Graph Problems. Theoretical Computer Science (TCS), 237-267, 1976.

Vertex Cover

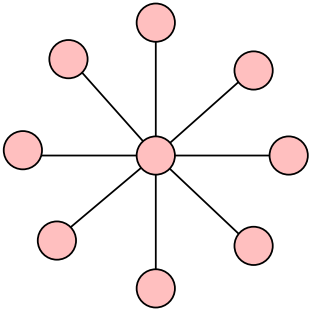
Vertex Cover: Given a graph $G = (V, E)$, find a subset of nodes $S \subseteq V$ of minimal cardinality such that for each edge $\{u, v\} \in E$, either u or v (or both) are in S.

Neighbor relation: $S - S'$ if S' can be obtained from S by adding or deleting a single node. Each vertex cover S has at most n neighbors.

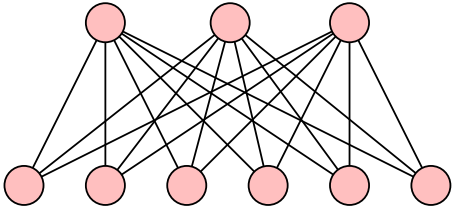
> *Gradient descent:* Start with $S = V$. If there is a neighbor S' that is a vertex cover and has lower cardinality, replace S with S'.

Remark: Algorithm terminates after at most n steps since each update decreases the size of the cover by one.

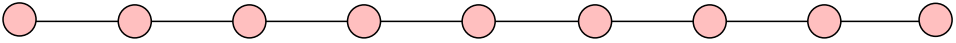
Vertex Cover



optimum: center node only
local optimum: all other nodes



optimum: all nodes at top
local optimum: all nodes at bottom



optimum: even nodes
local optimum: omit every third node

Besprochene Entwurfsmethoden:

- Brute-Force
- Fest-Parameter
- Divide-and-Conquer
- Greedy
- Dynamisches Programmieren
- lokale Suche

Es gibt weitere Entwurfsmethoden wie z.B.

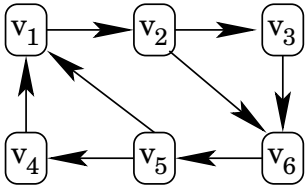
- ganzzahlige Programmierung (siehe „Wissensbasierte Systeme“)
- randomisierte Algorithmen
- approximative Algorithmen
- Backtracking (siehe ALD im Bachelorstudium)
- Branch-and-Bound (siehe ALD im Bachelorstudium)

Übersicht

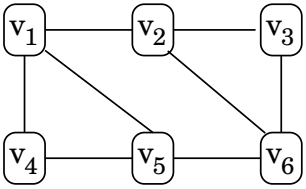
Ein *gerichteter Graph* $G = (V, E)$ besteht aus

- einer endlichen Menge von *Knoten* (engl. vertices) $V = \{v_1, \ldots, v_n\}$ und
- einer Menge von gerichteten *Kanten* (engl. edges) $E \subseteq V \times V$. [9]

Bei einem *ungerichteten Graphen* $G = (V, E)$ sind die Kanten ungeordnete Paare: $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$



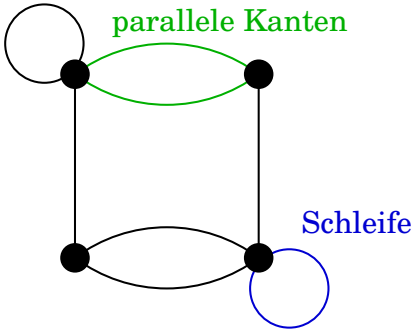
gerichteter Graph



ungerichteter Graph

[9] Zur Erinnerung: Das kartesische Produkt $V \times V$ ist die Menge aller geordneten Paare (a, b) mit $a \in V$ und $b \in V$ und beschreibt in diesem Fall die Menge aller möglichen Kanten mit Start- und Endknoten in V.

Übung 31. *Sehen Sie sich unsere Definition von ungerichteten Graphen noch einmal genau an. Überlegen Sie, ob parallele Kanten oder Schleifen möglich sind.*



Wie viele Kanten kann ein Graph maximal haben?

Motivation

Die Kanten eines Graphen können gewichtet sein, um z.B. Längen oder Zeiten beschreiben zu können.

Gewichtete Graphen G werden wir als 3-Tupel $G = (V, E, c)$ angeben, wobei

- V die Knotenmenge,
- E die Kantenmenge und
- $c : E \to \mathbb{R}$ eine Funktion bezeichnet, die jeder Kante e eine Zahl $c(e)$ zuordnet.

Gewichtete Graphen $G = (V, E, c)$ mit $c : E \to \mathbb{R}_0^+$, bei denen also keine negativen Kantengewichte vorkommen, nennen wir *Distanzgraphen*.

Motivation

Graphen verwenden wir überall dort, wo ein Sachverhalt darstellbar ist durch

- eine Menge von Objekten (Entitäten)
- und Beziehungen zwischen den Objekten.

Beispiele:

- *Routenplanung:* Städte sind durch Straßen verbunden; die Straßen haben eine gewisse Länge, die als Kantengewicht angegeben wird.
- *Kursplanung:* Kurse setzen andere Kurse voraus.
- *Produktionsplanung:* Produkte werden aus Einzelteilen und Teilprodukten zusammengesetzt.
- *Schaltkreisanalyse:* Bauteile sind durch elektrische Leitungen verbunden.
- *Spiele:* Der Zustand/Status eines Spiels wird durch einen Spielzug geändert.

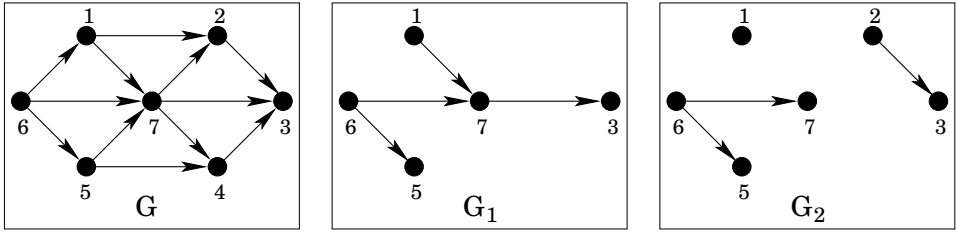
Sei $G = (V, E)$ ein gerichteter Graph und seien $u, v \in V$.

- Sei $e = (u, v)$ eine Kante. Knoten u ist der *Startknoten*, v der *Endknoten* von e. Die Knoten u und v sind *adjazent*, Knoten u bzw. v und Kante e sind *inzident*.
- Der *Eingangsgrad* von u, geschrieben $indeg(u)$, ist die Anzahl der in u einlaufenden Kanten.

  Der *Ausgangsgrad* von u ist die Anzahl der aus u auslaufenden Kanten und wird mit $outdeg(u)$ bezeichnet.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein *gerichteter Weg* in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.
- Der gerichteter Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.
- Ein gerichteter, einfacher Weg $p = (v_0, v_1, \ldots, v_k)$ heißt *Kreis*, falls $(v_k, v_0) \in E$ gilt und alle Kanten (v_{i-1}, v_i) und (v_k, v_0) paarweise disjunkt sind.
- Zwei Wege $P = (u_0, \ldots, u_k)$ und $P' = (v_0, \ldots, v_\ell)$ sind *knotendisjunkt*, falls $u_i \neq v_j$ für alle i und j mit $0 < i < k$ und $0 < j < \ell$ gilt. Die Endknoten der Wege werden bei dieser Ungleichheit nicht mit einbezogen.

Sei $G = (V, E)$ ein gerichteter Graph. Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$ gilt.

Beispiel: Die Graphen G_1 und G_2 sind Teilgraphen von G.



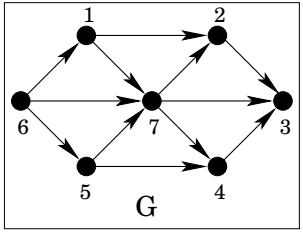
Für einen induzierten Teilgraphen G' von G fordern wir zusätzlich, dass eine Kante $e = (u, v) \in E$ mit $u, v \in V'$, bei der also Start- und Endknoten in V' liegen, auch im Teilgraphen G' vorhanden ist.

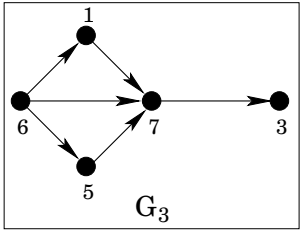
Sei $G = (V, E)$ ein gerichteter Graph. Ein Graph $G' = (V', E')$ heißt *induzierter Teilgraph* von G, falls $V' \subseteq V$ und $E' = E \cap (V' \times V')$ gilt. Dabei ist $V' \times V'$ die Menge aller möglichen Kanten mit Start- und Endknoten in V'.

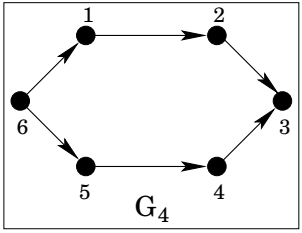
Der Graph $G|_{V'}$ bezeichnet den durch V' induzierten Teilgraphen von G. Wird in der Literatur auch als $G[V']$ geschrieben.

Oben: G_1 und G_2 sind keine induzierten Teilgraphen von G, da in G_1 die Kanten $(6, 1)$ und $(5, 7)$ fehlen und in G_2 bspw. die Kanten $(1, 7)$ und $(7, 2)$ fehlen.

Unten: G_3 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G und G_4 ist der durch die Knotenmenge $\{1, 2, 3, 4, 5, 6\}$ induzierte Teilgraph von G.







Sei $G = (V, E)$ ein ungerichteter Graph und seien $u, v \in V$.

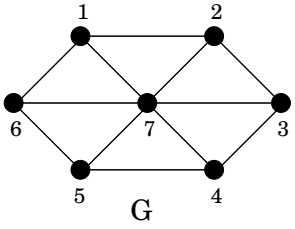
- Sei $e = \{u, v\}$ eine Kante. Die Knoten u und v sind *adjazent*, Knoten u bzw. v und Kante e sind *inzident*. Knoten u und v sind die *Endknoten* der Kante e.
- Der *Knotengrad* von u, geschrieben $deg(u)$, ist die Anzahl der zu u inzidenten Kanten.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein *ungerichteter Weg* in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 \leq i \leq k$.
- Der ungerichteter Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.
- Ein ungerichteter, einfacher Weg $p = (v_0, v_1, \ldots, v_k)$ heißt *Kreis*, falls $\{v_k, v_0\} \in E$ gilt und alle Kanten, also $\{v_{i-1}, v_i\}$ und $\{v_k, v_0\}$, paarweise disjunkt sind.
- Ein ungerichteter Graph heißt *azyklisch*, wenn er keinen Kreis enthält (kreisfrei).

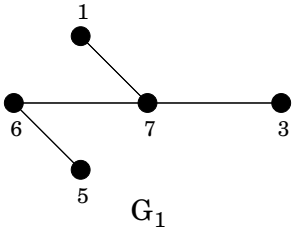
Sei $G = (V, E)$ ein ungerichteter Graph.

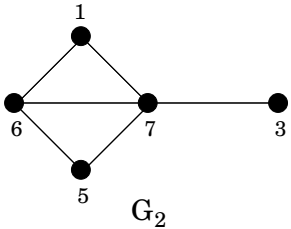
- Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.

  Unten: G_1 und G_2 sind Teilgraphen von G.
- Ein Graph $G' = (V', E')$ heißt *induzierter Teilgraph* von G, falls $V' \subseteq V$ und $E' = E \cap \{\{u, v\} \mid u, v \in V', u \neq v\}$ gilt.

Unten: G_1 ist kein induzierter Teilgraph von G, da die Kanten $\{1, 6\}$ und $\{5, 7\}$ fehlen. G_2 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G.





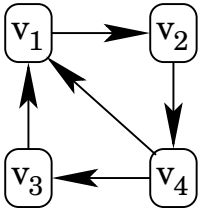


Speicherung von Graphen

Im einfachsten Fall wird ein Graph durch eine Liste der Kanten gespeichert.

- Speicherplatz: $\mathcal{O}(\mathcal{E})$
- Hinzufügen einer Kante: $\mathcal{O}(1)$
- Test, ob u adjazent zu v ist: $\mathcal{O}(\mathcal{E})$
- Zähle alle zu Knoten u benachbarten Knoten auf: $\mathcal{O}(\mathcal{E})$

Beispiel:

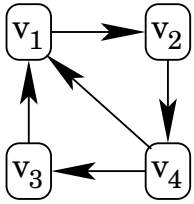


gerichtet: $(v_1, v_2), (v_2, v_4), (v_4, v_1), (v_4, v_3), (v_3, v_1)$
ungerichtet: $\{v_1, v_2\}, \{v_2, v_4\}, \{v_4, v_1\}, \{v_4, v_3\}, \{v_3, v_1\}$

Die *Adjazenz-Matrix* für G ist eine $\mathcal{V} \times \mathcal{V}$-Matrix $A_G = (a_{ij})$:

$$a_{ij} = \begin{cases} 0, & \text{falls } (v_i, v_j) \notin E \\ 1, & \text{falls } (v_i, v_j) \in E \end{cases}$$

Beispiel:



gerichtet:

A	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	0
4	1	0	1	0

ungerichtet:

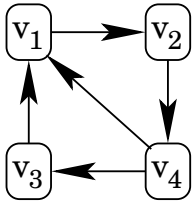
A	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

- Speicherplatz: $\mathcal{O}(\mathcal{V}^2)$
- Test, ob u adjazent zu v ist: $\mathcal{O}(1)$
- Zähle alle zu Knoten u benachbarten Knoten auf: $\mathcal{O}(\mathcal{V})$

Hinzufügen und Löschen von Knoten und Kanten ist möglich.

Bei einer *Adjazenz-Liste* werden für jeden Knoten v eines Graphen $G = (V, E)$ in einer doppelt verketteten Liste $Adj[v]$ alle von v ausgehenden Kanten gespeichert.

Beispiel:



gerichtet:

$Adj[v_1] = v_2$
$Adj[v_2] = v_4$
$Adj[v_3] = v_1$
$Adj[v_4] = v_1 \longleftrightarrow v_3$

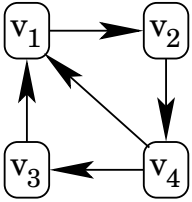
ungerichtet:

$Adj[v_1] = v_2 \longleftrightarrow v_3 \longleftrightarrow v_4$
$Adj[v_2] = v_1 \longleftrightarrow v_4$
$Adj[v_3] = v_1 \longleftrightarrow v_4$
$Adj[v_4] = v_1 \longleftrightarrow v_2 \longleftrightarrow v_3$

- Speicherplatz: $\mathcal{O}(\mathcal{V} + \mathcal{E})$
- Test, ob u adjazent zu v ist: $\mathcal{O}(deg(u))$
- Zähle alle zu Knoten u benachbarten Knoten auf: $\mathcal{O}(deg(u))$

Hinzufügen und Löschen von Knoten und Kanten ist einfach möglich.

Bei einem *Adjazenz-Array* werden alle Kanten $\{u, v\}$ (ungerichtet) bzw. (u, v) (gerichtet) in einem Array abgelegt. Alle zu einem Knoten inzidenten Kanten liegen hintereinander im Array.



gerichtet:

i	1	2	3	4	5
edgeOf	0	1	2	3	5

i	0	1	2	3	4	5
target	2	4	1	1	3	-

ungerichtet:

i	1	2	3	4	5
edgeOf	0	3	5	7	10

i	0	1	2	3	4	5	6	7	8	9	10
target	2	3	4	1	4	1	4	1	2	3	-

Im zweiten Array kann zusätzlich zum Zielknoten v auch das Gewicht der Kante gespeichert werden. Hinzufügen und Löschen von Knoten oder Kanten wird nicht unterstützt; nur geeignet für statische Graphen.

Vergleich:

Daten-struktur	Speicher-platz	Kante hinzufügen	ist Knoten v adjazent zu u?	iteriere über die Nachbarn von u
Kantenliste	$\mathcal{O}(\mathcal{E})$	$\mathcal{O}(1)$	$\mathcal{O}(\mathcal{E})$	$\mathcal{O}(\mathcal{E})$
Adjazenz-Matrix	$\mathcal{O}(\mathcal{V}^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\mathcal{V})$
Adjazenz-Liste	$\mathcal{O}(\mathcal{V}+\mathcal{E})$	$\mathcal{O}(1)$	$\mathcal{O}(deg(u))$	$\mathcal{O}(deg(u))$
Adjazenz-Array	$\mathcal{O}(\mathcal{V}+\mathcal{E})$	–	$\mathcal{O}(deg(u))$	$\mathcal{O}(deg(u))$

Die Adjazenz-Matrix ist geeignet für *dichte* Graphen (dense graphs), die anderen Datenstrukturen sind auch geeignet für *dünn besetzte* Graphen (sparse graphs).

Anmerkung: In der numerischen Mathematik müssen oft dünn besetzte Matrizen abgespeichert werden, also Matrizen, bei denen sehr viele Einträge gleich Null sind. Dort wird das „compressed row storage“[(10)] (CRS) oder auch „compressed sparse row“ (CSR) genannte Verfahren genutzt, das den Adjazenz-Arrays sehr ähnlich ist.

[(10)] `https://de.wikipedia.org/wiki/Compressed_Row_Storage`

Übersicht

Durchsuchen von gerichteten Graphen

Aufgabe: Die Suche soll alle von einem gegebenen Startknoten s aus erreichbaren Knoten finden.

Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

```
markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle aus s auslaufenden Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante (u, v) aus D
    falls der Knoten v als „unbesucht“ markiert ist:
        markiere Knoten v als „besucht“
        füge alle aus v auslaufenden Kanten zu D hinzu
```

Anmerkung: In der Datenstruktur D speichern wir diejenigen Kanten, von deren Endknoten vielleicht noch unbesuchte Knoten erreicht werden können.

Laufzeit:

- Jede Kante wird höchstens einmal in D eingefügt.
- Jeder Knoten wird höchstens einmal inspiziert.

$\Rightarrow$ Die Laufzeit ist proportional zur Anzahl der vom Startknoten aus erreichbaren Knoten und Kanten, also $\mathcal{O}(\mathcal{V} + \mathcal{E})$.

Der Typ der Datenstruktur bestimmt die *Durchlaufordnung:*

- Stack (last in, first out): *Tiefensuche*
- Liste (first in, first out): *Breitensuche*

Wir unterscheiden die Kanten bei einem gerichteten Graphen nach der Rolle, die sie bei der Tiefensuche spielen.

- *Baumkante:* Kante $(u, v) \in E$, der die Tiefensuche folgt, wo also während der Abarbeitung von `dfs(`u`)` ein Aufruf `dfs(`v`)` erfolgt.
- *Vorwärtskante:* Kante $(u, v) \in E$ mit dfb$[v] >$ dfb$[u]$, die keine Baumkante ist.
- *Querkante:* Eine Kante $(u, v) \in E$ mit dfb$[v] <$ dfb$[u]$ und dfe$[v] <$ dfe$[u]$.
- *Rückwärtskante:* Kante $(u, v) \in E$ mit dfb$[v] <$ dfb$[u]$ und dfe$[v] >$ dfe$[u]$.

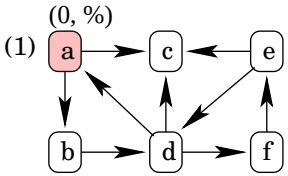
Rekursive Tiefensuche:
Sei $s \in V$ ein beliebiger Knoten.

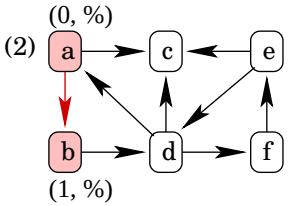
```
markiere alle Knoten als „unbesucht“
dfbZähler := 0
dfeZähler := 0
dfs(s)
```

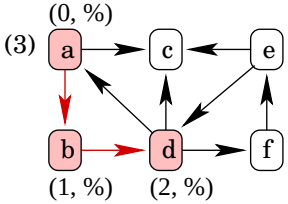
Diese Art der Tiefensuche, wo die Suche bei einem gegebenen Startknoten beginnt, bezeichnen wir als einfache Tiefensuche.

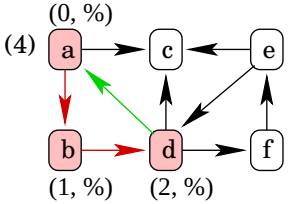
`dfs(`u`)`

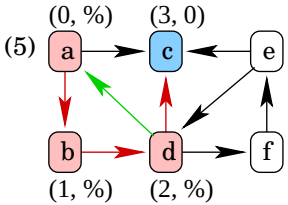
markiere u als „besucht“
dfb[u] := dfbZähler
dfbZähler := dfbZähler + 1
betrachte alle Kanten $(u, v) \in E$:
    falls Knoten v als „unbesucht“ markiert ist:
        `dfs(`v`)`
dfe[u] := dfeZähler
dfeZähler := dfeZähler + 1

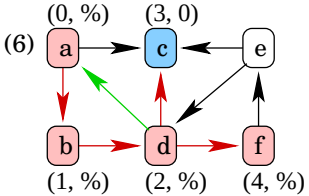

(1)
(0, %)
a
c
e
b
d
f

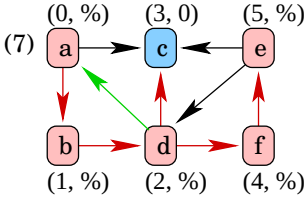

(2)
(0, %)
a
c
e
b
d
f
(1, %)

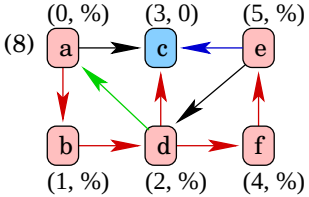

(3)
(0, %)
a
c
e
b
d
f
(1, %)
(2, %)

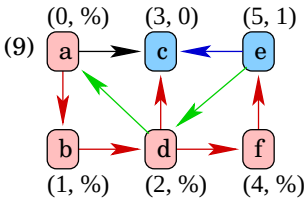

(4)
(0, %)
a
c
e
b
d
f
(1, %)
(2, %)

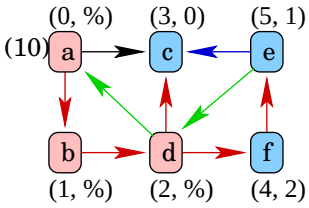

(5)
(0, %)
(3, 0)
a
c
e
b
d
f
(1, %)
(2, %)

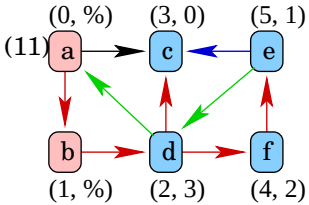

(6)
(0, %)
(3, 0)
a
c
e
b
d
f
(1, %)
(2, %)
(4, %)

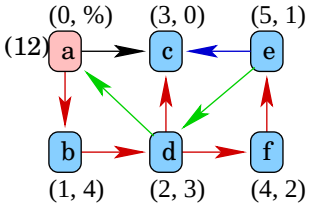

(7)
(0, %)
(3, 0)
(5, %)
a
c
e
b
d
f
(1, %)
(2, %)
(4, %)

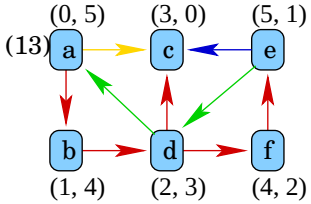

(8)
(0, %)
(3, 0)
(5, %)
a
c
e
b
d
f
(1, %)
(2, %)
(4, %)


(9)
(0, %)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, %)
(2, %)
(4, %)


(10)
(0, %)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, %)
(2, %)
(4, 2)

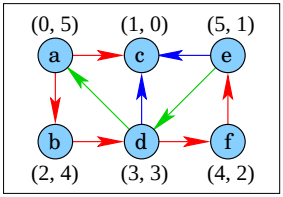

(11)
(0, %)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, %)
(2, 3)
(4, 2)

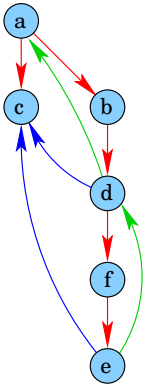

(12)
(0, %)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, 4)
(2, 3)
(4, 2)

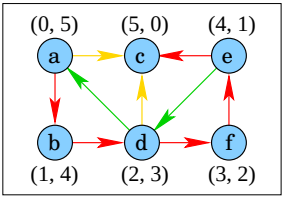


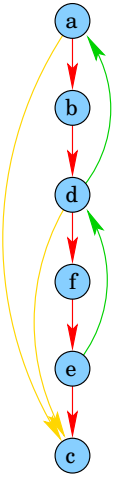
- Baumkanten sind rot markiert.
- Rückwärtskanten sind grün markiert.
- Querkanten sind blau markiert.
- Vorwärtskanten sind gelb markiert.

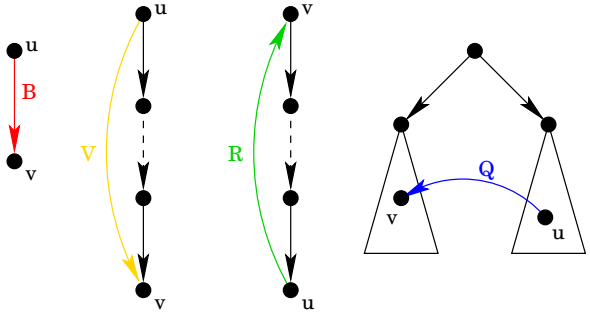
Werden die Knoten des Graphen in einer anderen Reihenfolge besucht, so ergibt sich eine andere Aufteilung der Kantenarten:











- Baumkante (u, v): $\text{dfb}[u] < \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$
- Vorwärtskante (u, v): $\text{dfb}[u] < \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$
- Rückwärtskante (u, v): $\text{dfb}[u] > \text{dfb}[v]$ und $\text{dfe}[u] < \text{dfe}[v]$
- Querkante (u, v): $\text{dfb}[u] > \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$

Übung 32. *Zeigen Sie, dass die Tiefensuche korrekt ist, dass also tatsächlich alle vom Startknoten s aus erreichbaren Knoten berichtet bzw. ausgegeben werden.*

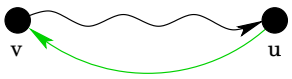
Tiefensuche: (Suche beginnt nicht bei einem ausgezeichneten Startknoten.)

```
markiere alle Knoten als unbesucht
solange ein unbesuchter Knoten v existiert:
    dfs(v)
```

Satz: Der gerichtete Graph G enthält einen Kreis $\iff$ die Tiefensuche auf G liefert eine Rückwärtskante.

Beweis:

"$\Leftarrow$" Für eine Rückwärtskante (u, v) gilt: $dfb(u) > dfb(v)$ und $dfe(u) < dfe(v)$. Außerdem existiert ein Weg von v nach u.



Gäbe es keinen Weg von v nach u, dann wäre `dfs(v)` beendet, bevor `dfs(u)` aufgerufen wird, also $dfe(v) < dfe(u)$. ↯↯

Beweis: (Fortsetzung)

"$\Rightarrow$" Sei $C = (v_1, v_2, \ldots, v_k, v_1)$ ein Kreis in G.

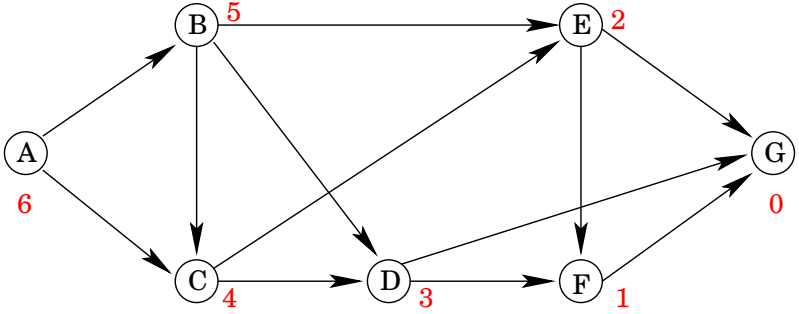
O.B.d.A. sei v_1 der Knoten aus C, der von der Tiefensuche zuerst besucht wird, also $dfb(v_1) < dfb(v_k)$.

`dfs(`v_1`)` wird erst beendet, wenn alle von v_1 aus erreichbaren Knoten, also insbesondere v_k, besucht und abgearbeitet wurden. Daher gilt $dfe(v_1) > dfe(v_k)$.

Also ist (v_k, v_1) eine Rückwärtskante.

Gegeben: Ein gerichteter Graph $G = (V, E)$.

Gesucht: Eine Nummerierung $\pi(v_1), \ldots, \pi(v_n)$ der Knoten, so dass gilt:
$(u, v) \in E \;\Rightarrow\; \pi(u) > \pi(v)$



Algorithmus: Tiefensuche

G ist kreisfrei $\Rightarrow$ dfe-Nummern sind topologische Sortierung!

Korrektheit: Für alle Kanten $(u, v) \in E$ muss $dfe(u) > dfe(v)$ gelten.

- Wenn G kreisfrei ist, treten keine Rückwärtskanten auf.
- (u, v) ist eine Baumkante
  $\rightarrow dfe(u) > dfe(v)$ ✓
  Denn: `dfs(u)` wird erst beendet, wenn `dfs(v)` bereits abgeschlossen ist.

- (u, v) ist eine Vorwärtskante
  $\rightarrow dfe(u) > dfe(v)$ ✓
  Denn: `dfs(v)` ist bereits beendet, während bei `dfs(u)` die Vorwärtskante entdeckt wird.

- (u, v) ist eine Querkante
  $\rightarrow dfe(u) > dfe(v)$ nach Definition ✓

Anmerkung: Wenn G nicht kreisfrei ist, also einen Kreis enthält, dann existiert keine topologische Sortierung der Knoten.

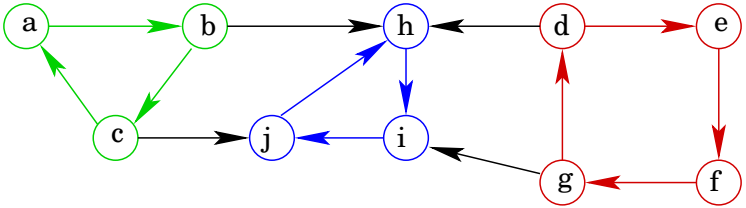
Übersicht

Sei $G = (V, E)$ ein gerichteter Graph.

- G ist *stark zusammenhängend*, wenn es zwischen jedem Knotenpaar einen Weg in G gibt.

  Also: Für alle $u, v \in V$ existiert ein Weg von u nach v *und* ein Weg von v nach u in G.
- Eine *starke Zusammenhangskomponente* von G ist ein bzgl. der Knotenmenge maximaler, stark zusammenhängender, induzierter Teilgraph von G.

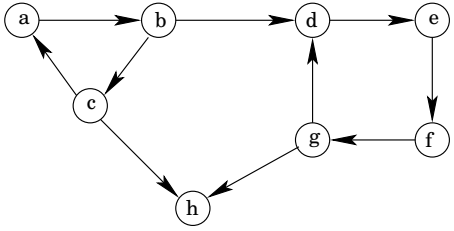
Beispiel:



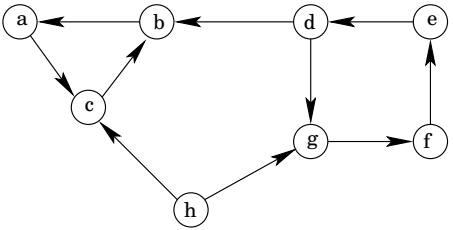
Algorithmus: Gegeben sei ein gerichteter Graph $G = (V, E)$.

1. Tiefensuche auf G mit dfe-Nummerierung der Knoten.
2. Berechne $G^R = (V, E^R)$ mit $E^R = \{(v, u) \mid (u, v) \in E\}$.
3. Tiefensuche auf G^R, wobei immer mit dem Knoten v mit *größtem dfe-Index aus Schritt 1* gestartet wird.

G

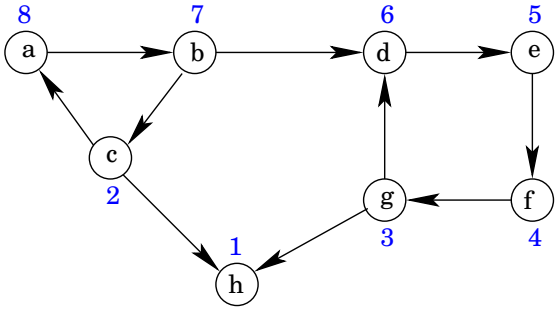


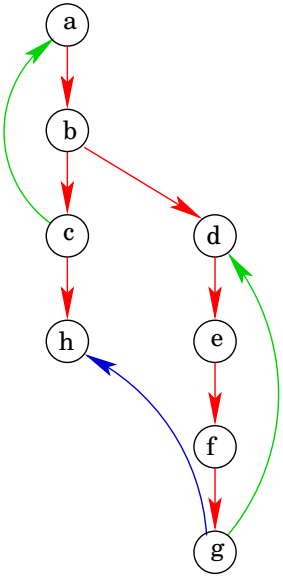
G^R



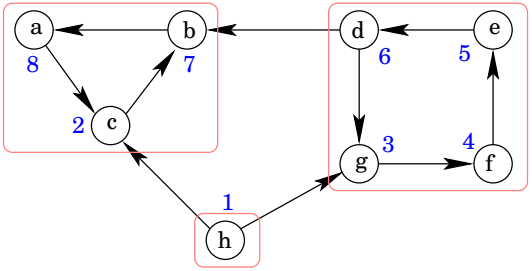
Jeder Baum des in Schritt 3 entstandenen Waldes entspricht einer starken Zusammenhangskomponente.

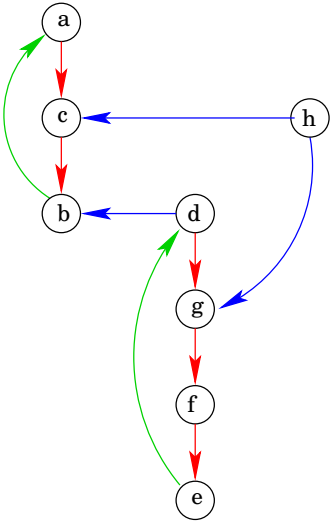
Beispiel: Eine mögliche Tiefensuche auf G mit Startknoten a liefert:



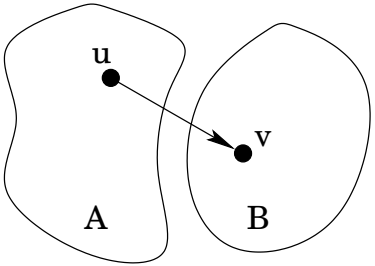


Beispiel: Tiefensuche auf G^R, wobei immer mit dem Knoten v mit *größtem dfe-Index aus dem vorigen Schritt* gestartet wird, liefert:





Warum ist es wichtig, in Schritt 3 den Knoten mit dem größten dfe-Index zu wählen?



Wenn eine Kante $(u, v) \in E$ existiert, die eine starke ZHK A mit einer starken ZHK B verbindet, dann gilt nach dem ersten Schritt: $dfe(u) > dfe(v)$
Eine Kante von B nach A kann also nicht existieren, da sonst A und B eine einzige starke ZHK bilden würden.

- falls die Tiefensuche in B beginnt:
  - → Tiefensuche in B ist beendet, bevor Tiefensuche in A startet.
  - → $dfe(u) > dfe(v)$
- falls die Tiefensuche in A beginnt:
  - → Die Tiefensuche erreicht irgendwann B, und B wird komplett abgearbeitet, bevor die Tiefensuche in A fortgesetzt wird.
  - → $dfe(u) > dfe(v)$

Korrektheit:

Wir stellen zunächst fest, dass die Graphen G und G^R dieselben starken Zusammenhangskomponenten haben, denn:

- $u \overset{G}{\rightsquigarrow} v \;\Rightarrow\; v \overset{G^R}{\rightsquigarrow} u$
  Wenn ein Weg von u nach v in G existiert, dann existiert in G^R ein Weg von v nach u.
- $v \overset{G}{\rightsquigarrow} u \;\Rightarrow\; u \overset{G^R}{\rightsquigarrow} v$
  Wenn ein Weg von v nach u in G existiert, dann existiert in G^R ein Weg von u nach v.

Korrektheit: (Fortsetzung)

Wir bestimmen die starken Zshg.-komponenten durch Tiefensuche auf G^R. Betrachten wir zwei starke Zshg.-komponenten A und B:

- Eine Kante $(u, v) \in E$ von A nach B wird in G^R umgedreht.
- In G kann keine Kante von B nach A existieren, da sonst A und B eine große Zshg.-komponente bilden würde.
- Es existieren in G^R also keine Kanten von A nach B.
- Zunächst wird eine Tiefensuche in A gestartet, da aufgrund unserer Vorüberlegung $dfe(u) > dfe(v)$ gilt.

$\rightarrow$ Der Tiefensuchbaum umfasst genau die Knoten in A.

Anschließend sind die Knoten von A als besucht markiert und werden nie wieder besucht. Also werden auch die Knoten von B durch eine spätere Tiefensuche korrekt ausgegeben.

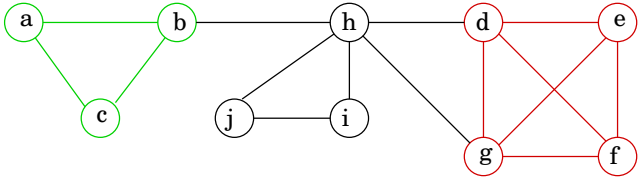
Sei $G = (V, E)$ ein ungerichteter Graph.

- G heißt *zusammenhängend*, wenn es zwischen jedem Knotenpaar $u, v \in V$ einen Weg in G gibt.
- G heißt *k-fach zusammenhängend*, wenn es zwischen jedem Knotenpaar k viele knotendisjunkte Wege gibt. Das heißt, außer die Start- und Endknoten sind alle auf den Wegen liegenden Knoten paarweise verschieden.

  Oder: Nach dem Entfernen von $k - 1$ vielen Knoten ist der verbleibende Graph immer noch zusammenhängend.
- Eine *k-fache Zusammenhangskomponente* von G ist ein maximaler k-fach zusammenhängender, induzierter Teilgraph von G.
- Ein Knoten u ist ein *Schnittpunkt*, wenn der Graph G ohne Knoten u (und damit auch ohne die zu u inzidenten Kanten) mehr Zusammenhangskomponenten hat als G.

  Englisch: *cut point* oder *articulation point*.

Beispiel: Der folgende Graph ist zusammenhängend.

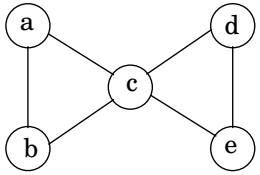


- Eine der 2-fach Zusammenhangskomponenten ist der grün markierte Teilgraph.
- Der rot markierte Teilgraph ist sogar eine 3-fach Zusammenhangskomponente.
- Knoten b und h sind Schnittpunkte: Wird einer dieser Knoten entfernt, ist der Restgraph nicht mehr zusammenhängend.

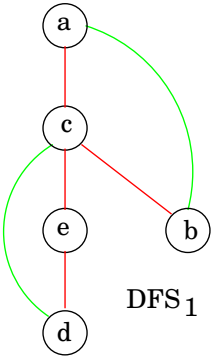
Wir werden im Folgenden solche Schnittpunkte berechnen. Ein Graph ohne Schnittpunkt ist mindestens 2-fach zusammenhängend.

Übung 33. *Sei $G = (V, E)$ ein ungerichteter Graph. Zeigen Sie: Wenn G nicht zusammenhängend ist, dann ist der komplementäre Graph $\overline{G} = (V, \overline{E})$ mit der Kantenmenge $\overline{E} := \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$ zusammenhängend.*

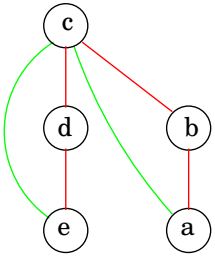
Tiefensuche auf ungerichteten Graphen G:



Graph G



DFS_1



DFS_2

(a) Es existieren keine Querkanten.

(b) Die Wurzel w eines DFS-Baums ist ein Schnittpunkt, wenn w mehr als einen Teilbaum hat, denn: G ohne w zerfällt wegen Punkt (a) in mehrere Teile, siehe Knoten c bei DFS_2.

(c) Ein innerer Knoten v ist *kein* Schnittpunkt, wenn aus jedem Teilbaum von v eine Rückwärtskante zu einem Vorgänger von v geht, denn: Wenn v aus G entfernt wird, existiert über die Rückwärtskante ein alternativer Weg in den Teilbaum.

Knoten c in DFS_1 ist ein Schnittpunkt: Vom rechten Teilbaum geht zwar eine Rückwärtskante zu einem Vorgänger von c im Baum, aber vom linken Teilbaum nicht.

Idee des Algorithmus: Tiefensuche auf G.

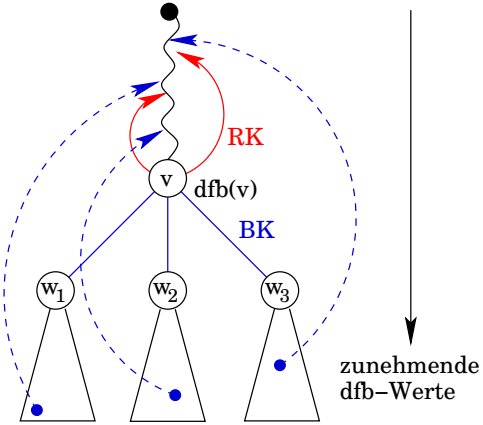
- Berechne für Knoten v am Ende von `dfs(`v`)`:

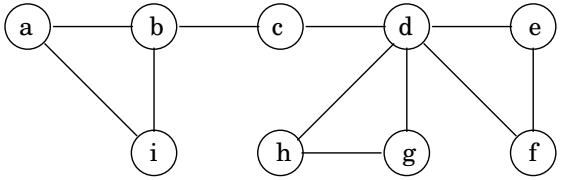
$$low(v) := \min\left\{ dfb(v), \min_{(v,z)\in RK}\{dfb(z)\}, \min_{(v,w)\in BK}\{low(w)\} \right\}$$

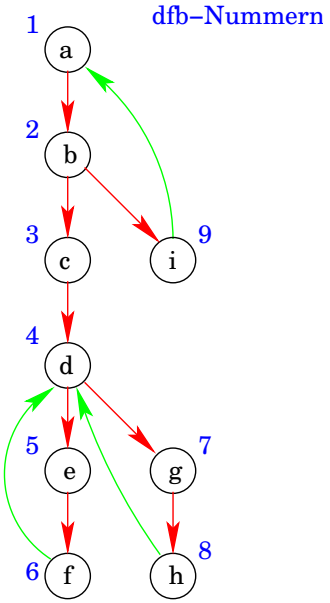
- „Wie hoch geht es im Baum über Rückwärtskanten?“

Anschaulich:

$$low(v) := \min \left\{ dfb(v), \min_{(v,z) \in RK} \{dfb(z)\}, \min_{(v,w) \in BK} \{low(w)\} \right\}$$







low(f) = min(6, 4, %) = 4

low(e) = min(5, %, 4) = 4

low(h) = min(8, 4, %) = 4

low(g) = min(7, %, 4) = 4

low(d) = min(4, %, 4, 4) = 4

low(c) = min(3, %, 4) = 3

low(i) = min(9, 1, %) = 1

low(b) = min(2, %, 1) = 1

low(a) = min(1, %, 1) = 1

Algorithmus:

markiere alle Knoten als „unbesucht“
dfbZähler := 1
initialisiere einen leeren Stack
für alle Knoten $v \in V$:
    falls v als „unbesucht“ markiert ist
        `ZSuche(`v`)`

Vorbemerkungen zur Funktion `ZSuche`:

- Zeile 10: Ein Knoten v ist ein Schnittpunkt, wenn ein Kind w von v existiert mit $low(w) \geq dfb(v)$
- Der Wert $low(v)$ wird in den Zeilen 4, 11 und 13 berechnet.
- Zeile 8: Der Vorgänger pred[v] wird vermerkt, um in Zeile 12 Rückwärts- von Baumkanten unterscheiden zu können.

ZSuche(v)

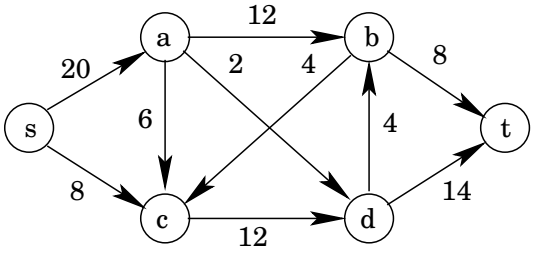
```
1. markiere v als „besucht“
2. dfb[v] := dfbZähler
3. dfbZähler := dfbZähler + 1
4. low[v] := dfb[v]
5. betrachte alle mit v inzidenten Kanten {v,w} ∈ E:
6.       lege {v,w} auf Stack, falls noch nicht geschehen
7.       falls w als „unbesucht“ markiert ist:
8.             pred[w] := v
9.             ZSuche(w)     // berechnet low[w]
10.            falls low[w] ≥ dfb[v]: alle Kanten bis {v,w} vom
                     Stack nehmen und als Komponente ausgeben
11.            low[v] := min(low[v], low[w])
12.      sonst: falls w ≠ pred[v]:
13.            low[v] := min(low[v],dfb[w])
```

Übersicht

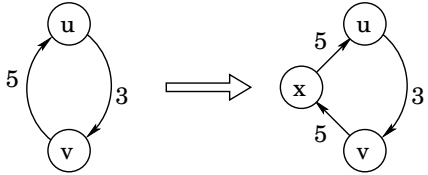
Gegeben:
- Ein gewichteter Graph $G = (V, E, c)$ ohne antiparallele Kanten mit einer Kostenfunktion $c : E \to \mathbb{Q}^+$.
- Eine Quelle $s \in V$ und eine Senke $t \in V$, wobei $s \neq t$ gilt.

Gesucht: Maximaler Fluss im Netzwerk von Quelle s nach Senke t.

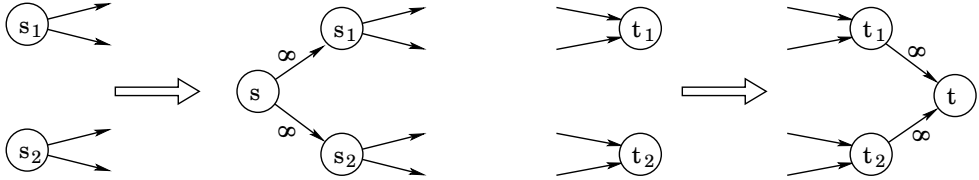
Beispiel:



Falls der gegebene Graph antiparallele Kanten besitzt, muss der Graph transformiert werden, indem neue Knoten eingefügt werden:



Sollen mehrere Quellen oder Senken modelliert werden, muss der Graph ebenfalls durch einfügen neuer Knoten modifiziert werden:



Eine *Flussfunktion* f eines Netzwerks $H = (V, E, c, \mathbb{Q}^+, s, t)$ ist eine Funktion $f : E \to \mathbb{Q}^+$ mit

- Kapazitätsbeschränkung: $\forall e \in E : \ 0 \leq f(e) \leq c(e)$
- Flusserhaltung: $\forall v \in V - \{s, t\} :$

$$\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$$

Der *Fluss* $F(f)$ im Netzwerk $H = (V, E, c, \mathbb{Q}^+, s, t)$ ist definiert als

$$F(f) = \sum_{e \in In(t)} f(e) - \sum_{e \in Out(t)} f(e)$$

oder analog:

$$F(f) = \sum_{e \in Out(s)} f(e) - \sum_{e \in In(s)} f(e)$$

Sei $H = (V, E, c, \mathbb{Q}^+, s, t)$ ein Netzwerk, und sei $S \subset V$ eine Teilmenge der Knoten mit $s \in S$ und $t \notin S$, sei $\overline{S} = V - S$. Dann definieren wir:

- $E(S, \overline{S}) := \{(x, y) \mid x \in S,\ y \in \overline{S}\} \cap E$
- $E(\overline{S}, S) := \{(x, y) \mid x \in \overline{S},\ y \in S\} \cap E$
- Der *Schnitt des Netzwerks* H bzgl. der Knotenmenge S ist $H(S) := E(S, \overline{S}) \cup E(\overline{S}, S)$.

  Also: In $H(S)$ sind all die Kanten, die aus der Menge S in die Menge $\overline{S}$ laufen und umgekehrt.
- Die *Kapazität des Schnitts* $H(S)$ ist definiert als

$$c(S) := \sum_{e \in E(S, \overline{S})} c(e).$$

  Achtung: Berücksichtigt nur Kanten von S nach $\overline{S}$.

Lemma:

$$F(f) = \sum_{e \in E(S,\overline{S})} f(e) - \sum_{e \in E(\overline{S},S)} f(e)$$

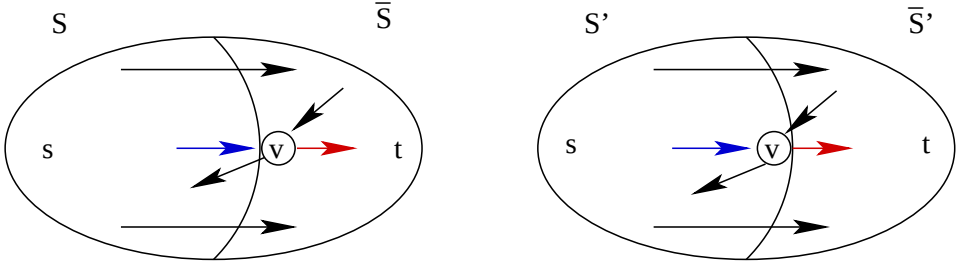
Beweis: Induktion über $|S|$.

Induktionsanfang: $S = \{s\}$

$$\begin{aligned} F(f) &\stackrel{!}{=} \sum_{E(S,\overline{S})} f(e) - \sum_{E(\overline{S},S)} f(e) \\ &= \sum_{e \in Out(s)} f(e) - \sum_{e \in In(s)} f(e) \end{aligned}$$

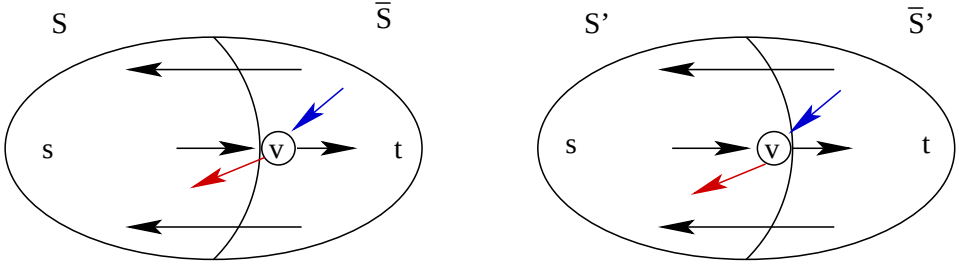
$\longrightarrow$ O.K.: Gilt nach Definition von Netzwerkfluss.

(1)



$$\sum_{E(S\cup\{v\},\overline{S}-\{v\})} f(e) = \sum_{E(S,\overline{S})} f(e) - \sum_{\substack{e=(u,v)\in E \\ u\in S}} f(e) + \sum_{\substack{e=(v,w)\in E \\ w\in\overline{S}}} f(e)$$

(2)



$$\sum_{E(\overline{S}-\{v\},S\cup\{v\})} f(e) = \sum_{E(\overline{S},S)} f(e) - \sum_{\substack{e=(v,w)\in E \\ w\in S}} f(e) + \sum_{\substack{e=(u,v)\in E \\ u\in\overline{S}}} f(e)$$

$$\sum_{E(S\cup\{v\},\overline{S}-\{v\})} f(e) = \sum_{E(S,\overline{S})} f(e) - \underbrace{\sum_{\substack{e=(u,v)\in E\\ u\in S}} f(e)}_{A} + \underbrace{\sum_{\substack{e=(v,w)\in E\\ w\in \overline{S}}} f(e)}_{B}$$

$$\sum_{E(\overline{S}-\{v\},S\cup\{v\})} f(e) = \sum_{E(\overline{S},S)} f(e) - \underbrace{\sum_{\substack{e=(v,w)\in E\\ w\in S}} f(e)}_{C} + \underbrace{\sum_{\substack{e=(u,v)\in E\\ u\in \overline{S}}} f(e)}_{D}$$

Subtrahieren wir die beiden Gleichungen, dann gilt:

$$\begin{aligned} F(f) &\stackrel{!}{=} \sum_{E(S\cup\{v\},\overline{S}-\{v\})} f(e) - \sum_{E(\overline{S}-\{v\},S\cup\{v\})} f(e) \\ &= \sum_{E(S,\overline{S})} f(e) - \sum_{E(\overline{S},S)} f(e) + (B+C) - (A+D) \end{aligned}$$

$$A + D = \sum_{\substack{e=(u,v)\in E \\ u\in S}} f(e) + \sum_{\substack{e=(u,v)\in E \\ u\in \overline{S}}} f(e) = \sum_{e\in In(v)} f(e)$$

$$B + C = \sum_{\substack{e=(v,w)\in E \\ w\in \overline{S}}} f(e) + \sum_{\substack{e=(v,w)\in E \\ w\in S}} f(e) = \sum_{e\in Out(v)} f(e)$$

Da für jeden Knoten $v \in V - \{s, t\}$ die Flusserhaltung

$$\sum_{e\in In(v)} f(e) = \sum_{e\in Out(v)} f(e)$$

gilt, erhalten wir $(B + C) - (A + D) = 0$.

Also: Der Fluss über den alten Schnitt ist gleich dem Fluss über den neuen Schnitt.

Satz: Für jede Teilmenge $S \subset V$ mit $s \in S$ und $t \notin S$ gilt:

$$F(f) \leq c(S)$$

Beweis:

$$\begin{aligned} F(f) &\overset{\text{Lemma}}{=} \sum_{E(S,\overline{S})} f(e) - \sum_{E(\overline{S},S)} f(e) \\ &\overset{0 \leq f(e) \leq c(e)}{\leq} \sum_{E(S,\overline{S})} c(e) - \sum_{E(\overline{S},S)} f(e) \\ &\overset{\text{Definition}}{=} c(S) - \sum_{E(\overline{S},S)} f(e) \\ &\overset{0 \leq f(e)}{\leq} c(S) \end{aligned}$$

Wegen $F(f) \leq c(S)$ gilt also insbesondere:

$$\max_f\{F(f)\} \;\leq\; \min_S\{c(S)\}$$

Damit erhalten wir das *Max-Flow Min-Cut Theorem*:

$$F(f) = c(S) \qquad \Longrightarrow \qquad F(f) \text{ ist maximal, } c(S) \text{ ist minimal}$$

Definition: 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* im Pfad P.

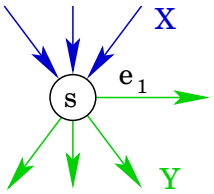
Idee der meisten Flussalgorithmen:

1. Starte mit beliebiger Flussfunktion f, z.B. $f(e) := 0 \; \forall e \in E$
2. Suche Pfad $P = u_1, \ldots, u_k$ mit $u_1 = s$, $u_k = t$ und
   - für alle Vorwärtskanten $e = (u_i, u_{i+1})$ gilt: $f(e) < c(e)$
     sei $\Delta(e) := c(e) - f(e)$
   - für alle Rückwärtskanten $e = (u_{i+1}, u_i)$ gilt: $f(e) > 0$
     sei $\Delta(e) := f(e)$

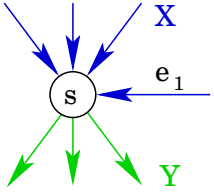
   Sei $\Delta(P)$ das kleinste $\Delta(e)$ über alle Kanten e im Pfad P.
3. Erhöhe den Fluss um $\Delta(P)$.
   - für eine Vorwärtskante e also: $f'(e) := f(e) + \Delta(P)$
   - für eine Rückwärtskante e also: $f'(e) := f(e) - \Delta(P)$
4. Wiederhole die Schritte 2 und 3 solange es einen Pfad P mit $\Delta(P) > 0$ gibt.

Aufgrund der Wahl von $\Delta(P)$ ist klar, dass in Schritt 3 die Kapazitätsbeschränkung pro Kante eingehalten wird.

Um zu zeigen, dass der Fluss vergrößert wurde, betrachten wir zwei Fälle. Sei e_1 die zu s inzidente Kante, deren Fluss geändert wurde.



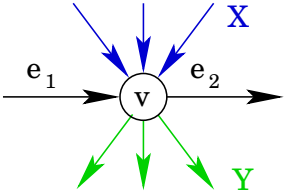
$$
\begin{aligned}
F(f') &= \sum_{e \in Out(s)} f'(e) - \sum_{e \in In(s)} f'(e) \\
&= \sum_{e \in Y} f(e) + f'(e_1) - \sum_{e \in X} f(e) \\
&= \sum_{e \in Y} f(e) + f(e_1) + \Delta(P) - \sum_{e \in X} f(e) \\
&= F(f) + \Delta(P) \; > \; F(f)
\end{aligned}
$$



$$
\begin{aligned}
F(f') &= \sum_{e \in Out(s)} f'(e) - \sum_{e \in In(s)} f'(e) \\
&= \sum_{e \in Y} f(e) - \Big(\sum_{e \in X} f(e) + f'(e_1) \Big) \\
&= \sum_{e \in Y} f(e) - \Big(\sum_{e \in X} f(e) + f(e_1) - \Delta(P) \Big) \\
&= F(f) + \Delta(P) \; > \; F(f)
\end{aligned}
$$

Bei der Flusserhaltung im Knoten v unterscheiden wir vier Fälle. Seien e_1 und e_2 die zu v inzidenten Kanten, deren Fluss geändert wurde.

Fall 1:

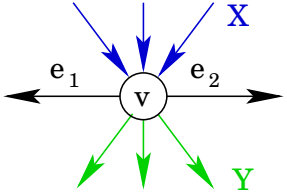


Über e_1 fließt zusätzlich $\Delta(P)$ in den Knoten v ein, was über e_2 aber auch wieder abfließt.

$$\sum_{e \in In(v)} f'(e) = \sum_{e \in In(v)} f(e) + \Delta(P)$$

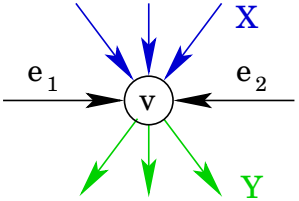
$$\sum_{e \in Out(v)} f'(e) = \sum_{e \in Out(v)} f(e) + \Delta(P)$$

Fall 2:



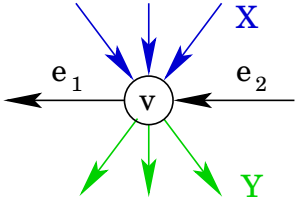
Bei der Kante e_1 wird der Fluss um $\Delta(P)$ verringert, bei der Kante e_2 um $\Delta(P)$ vergrößert, so dass in Summe bei den ausgehenden Kanten keine Änderung erfolgt.

Fall 3:



Bei der Kante e_1 wird der Fluss um $\Delta(P)$ vergrößert, bei der Kante e_2 um $\Delta(P)$ verringert, so dass in Summe bei den einlaufenden Kanten keine Änderung erfolgt.

Fall 4:



Über e_2 fließt zusätzlich $\Delta(P)$ in den Knoten v ein, was über e_1 aber auch wieder abfließt.

$$\sum_{e \in In(v)} f'(e) = \sum_{e \in In(v)} f(e) + \Delta(P)$$

$$\sum_{e \in Out(v)} f'(e) = \sum_{e \in Out(v)} f(e) + \Delta(P)$$

Ein Pfad P mit $\Delta(P) > 0$ ist ein *zunehmender Pfad*.

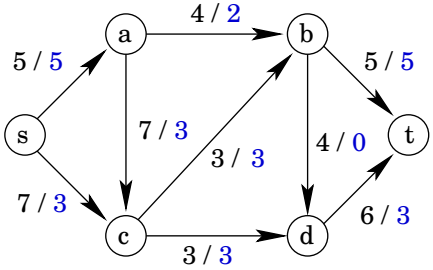
Sei $G_R(f) = (V, E_R)$ der *Restgraph* zu f, der alle noch möglichen Flussvergrößerungen beschreibt. Für jede Kante $e \in E$ von u nach v gibt es im Restgraphen

- eine Kante von u nach v mit Gewicht $c(e) - f(e)$, falls $c(e) > f(e)$,
- und eine Kante von v nach u mit Gewicht $f(e)$, falls $f(e) > 0$.

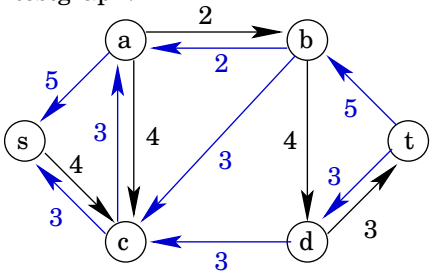
Bemerkung: Es müssen auch rückwärtsgerichtete Kanten betrachtet werden, damit eine Flussvergrößerung berechnet werden kann, siehe dazu im Beispiel auf der nächsten Folie den Graphen G. Es findet sich dort kein Weg, der nur Vorwärtskanten enthält, bei dem der Fluss vergrößert werden könnte.

Beispiel:

Graph:



Restgraph:



Bemerkung: Jeder Weg von s nach t im Restgraphen $G_R(f)$ ist ein zunehmender Pfad in G.

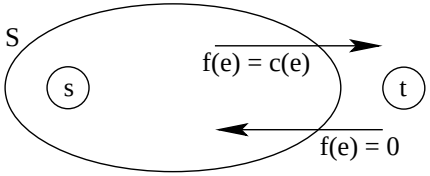
hier: $P = s \xrightarrow{4} c \xrightarrow{3} a \xrightarrow{2} b \xrightarrow{4} d \xrightarrow{3} t$ mit $\Delta(P) = 2$

Satz: $F(f)$ ist maximal $\iff$ in $G = (V, E)$ gibt es keinen zunehmenden Pfad

Beweis:

"$\Rightarrow$" Klar. Denn gäbe es einen zunehmenden Pfad, dann könnte $F(f)$ vergrößert werden und $F(f)$ wäre nicht maximal.

"$\Leftarrow$" Wenn es in G keinen zunehmenden Pfad gibt, dann gibt es in $G_R(f)$ keinen Weg von s nach t. Sei S die Menge der Knoten, die in $G_R(f)$ von s aus erreichbar sind, sei $\overline{S}$ die Menge der übrigen Knoten und betrachte den Graphen G:



Dann gilt für jede Kante

- $e \in E(S, \overline{S})$: $f(e) = c(e)$
- $e \in E(\overline{S}, S)$: $f(e) = 0$

Damit ist $F(f) = c(S)$, und somit ist der Fluss maximal.

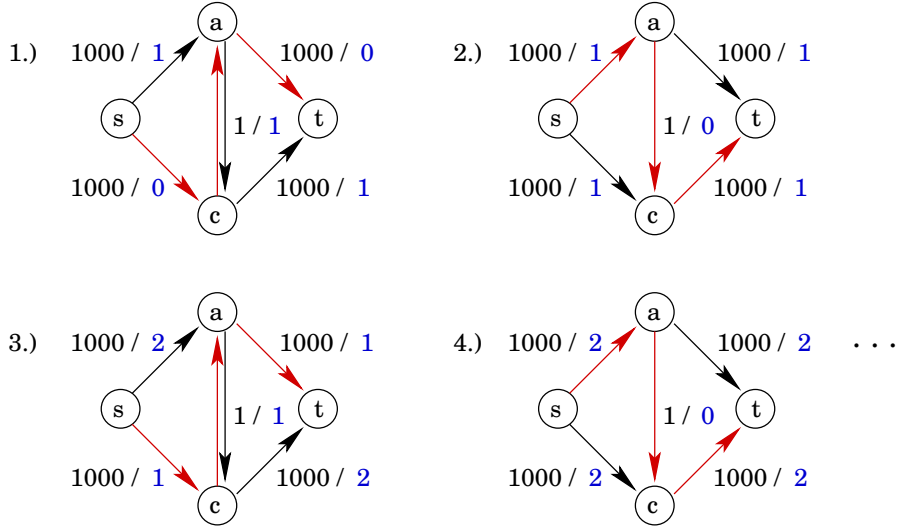
Idee von Ford-Fulkerson (1956): Suche einen beliebigen zunehmenden Pfad P (z.B. mittels Tiefensuche) und erhöhe den Fluss entlang des Pfades um $\Delta(P)$.

Bemerkungen:

- ohne Beweis[11]: Der Algorithmus terminiert nicht immer für irrationale Kapazitäten.
- ohne Beweis: Der Algorithmus konvergiert für irrationale Kapazitäten nicht einmal unbedingt gegen $F_{\max}$.
- Für ganzzahlige Kapazitäten ist die Anzahl der Flussvergrößerungen durch $F_{\max}$ beschränkt.

  Daher: Die Anzahl der Flussvergrößerungen ist abhängig von den Kantengewichten, und nicht allein abhängig von der Größe des Graphen!

[11] Uri Zwick: The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. Theoretical Computer Science (1995).



Frage: Ist die Laufzeit polynomiell in der Eingabegröße?

Idee: Wähle immer einen zunehmenden Pfad mit der minimalen Anzahl Kanten, wie ihn bspw. eine Breitensuche liefert.

Lemma: Der Netzwerkfluss-Algorithmus nach Edmonds und Karp benötigt höchstens $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ Iterationen.

Damit erhalten wir als *Laufzeit* insgesamt:

- Anzahl Iterationen: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$
- Breitensuche zum Finden kürzester Pfade: $\mathcal{O}(\mathcal{E})$
- Gesamte Laufzeit: $\mathcal{O}(\mathcal{E}^2 \cdot \mathcal{V})$

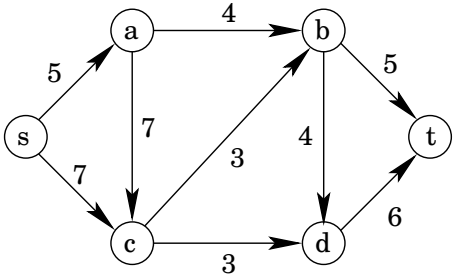
Beweis: Sei d die Distanz von der Quelle s zur Senke t im aktuellen Restgraphen. Wir werden zeigen, dass

- d niemals kleiner wird, und
- nach $\mathcal{O}(\mathcal{E})$ Iterationen der Wert von d um mindestens 1 wächst.

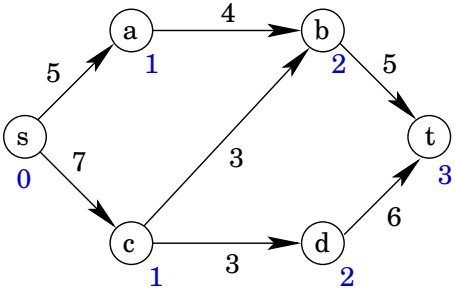
Der dem Restgraphen $G_R = (V, E_R)$ zugeordnete Schichtengraph $L(G_R)$ ist derjenige Teilgraph von G_R, der nur Kanten $(u, v) \in E_R$ enthält, für die $\delta(s, v) = \delta(s, u) + 1$ gilt.

$L(G_R)$ enthält für jeden Knoten $v \in V$ alle Pfade von s nach v minimaler Länge.

Restgraph:



Schichtengraph:



Für einen Pfad P minimaler Länge in $G_R(f)$ gilt also: Die Kanten von P liegen in $L(G_R(f))$.

Nun wird der Fluss entlang des Pfades P um $\Delta(P)$ erhöht. Der neue Restgraph $G_R(f')$ unterscheidet sich von $G_R(f)$ dadurch,

- dass mindestens eine Kante auf P gesättigt ist und daher in $G_R(f')$ nicht vorhanden ist.
- dass evtl. weitere, allerdings rückwärts gerichtete Kanten hinzugekommen sind.

Betrachten wir nun einen Pfad Q minimaler Lange in $G_R(f')$:

- Falls Q auch in $L(G_R(f))$ enthalten ist, dann muss Q dieselbe Länge wie P haben.
- Falls Q nicht in $L(G_R(f))$ enthalten ist, muss die Länge von Q um mindestens 1 größer sein als die Länge von P, da nur rückwärts gerichtete Kanten hinzugekommen sind.

Die Pfadlänge wird also nicht kleiner.

Betrachten wir eine Folge von Flussvergrößerungen.

- Ein Restgraph G_R enthält höchstens $2 \cdot \mathcal{E}$ Kanten, da zu jeder Kante des gegebenen Netzwerks G höchstens eine weitere, entgegengesetzt gerichtete Kante hinzukommt.
- Nach spätestens $\mathcal{O}(\mathcal{E})$ Iterationen ist jede Kante entfernt worden, so dass eine rückwärts gerichtete Kante genutzt werden muss und die Pfadlänge um mindestens 1 größer wird.
- Die Distanz von s nach t kann höchstens $\mathcal{V}$-mal erhöht werden, da ein kürzester Weg keine Knoten doppelt besucht, oder anders gesagt, die maximale Pfadlänge ist $\mathcal{V} - 1$.

Damit haben wir gezeigt, dass der Netzwerkfluss-Algorithmus nach Edmonds und Karp höchstens $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ Iterationen benötigt.

Idee von Dinic: Finde alle kürzesten Wege mit gleicher Anzahl von Kanten in einer Runde.

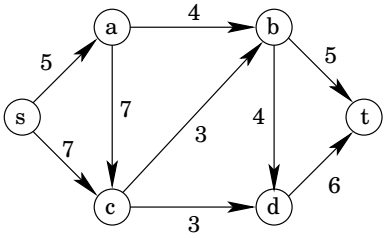
1. Bilde den Schichtengraphen $L(G_R(f))$. Dieser kann mittels Breitensuche in Zeit $\mathcal{O}(\mathcal{E})$ aufgebaut werden.
2. Bestimme eine Flussfunktion f' für den Schichtengraphen $L(G_R(f))$, die nicht durch einen zunehmenden Pfad von s nach t erhöhbar ist und addiere f' zu f hinzu.

   Mit anderen Worten: Jeder Weg von s nach t im Schichtengraphen enthält eine gesättigte Kante. Wir nennen f' einen blockierenden Fluss für den Schichtengraphen.
3. Wiederhole Schritt 1 und 2 solange, bis der Fluss f maximal ist, also bis es keinen Weg von s nach t im Schichtengraphen $L(G_R(f))$ gibt.

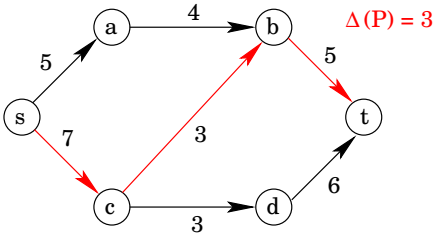
Aber wie findet man einen blockierenden Fluss?

Beispiel: Finden einer nicht-vergrößerbaren Flussfunktion.

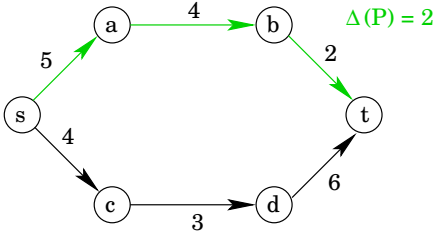
Restgraph:



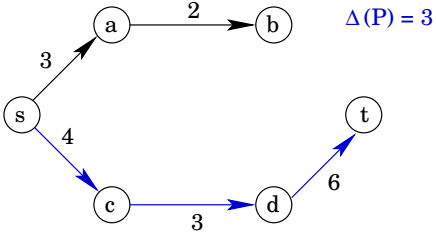
1. Schichtengraph:



2. Schichtengraph:



3. Schichtengraph:



Finden einer nicht-vergrößerbaren Flussfunktion:

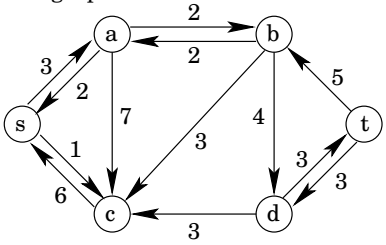
- Wir modifizieren die Tiefensuche so, dass die Suche endet, wenn der Knoten t erreicht wurde, oder wenn ein Knoten ohne auslaufende Kanten erreicht wurde.
  $\rightarrow$ Laufzeit: $\mathcal{O}(\mathcal{V})$
- Starte eine modifizierte Tiefensuche bei s:
  - Wenn der Knoten t durch die Tiefensuche erreicht wird, wurde ein zunehmender Pfad P von s nach t gefunden:
    - Verkleinere die Kapazitäten der Kanten auf P um $\Delta(P)$.
    - Mindestens eine Kante erhält die Kapazität 0: Entferne diese Kante aus dem Schichtengraph.
  - Wenn Knoten t nicht erreicht wird, entferne die letzte Kante des Weges, damit keine Endlosschleifen auftreten.

  $\rightarrow$ Laufzeit: $\mathcal{O}(\mathcal{V})$
- Starte die Tiefensuche erneut. $\rightarrow$ Da in jedem Durchlauf eine Kante entfernt wird, gibt es höchstens $\mathcal{O}(\mathcal{E})$ Iterationen.

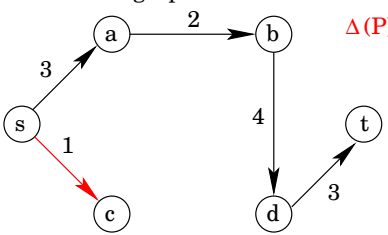
Laufzeit: $\mathcal{O}(\mathcal{E} \cdot \mathcal{V})$

weiter im *Beispiel:*

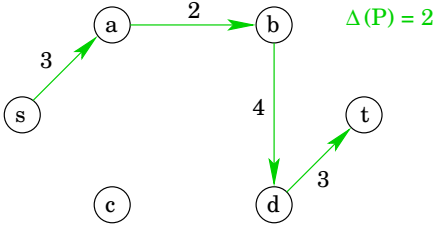
Restgraph:



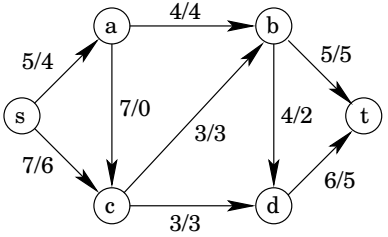
1. Schichtengraph:



2. Schichtengraph:



maximaler Fluss:



Laufzeit:

- Nach jeder Iteration der Schritte 1 und 2 des Algorithmus erhöht sich die Anzahl der Kanten in den kürzesten Wegen um mindestens 1.

  Da ein kürzester Weg keinen Knoten mehrfach besucht, gibt es höchstens $\mathcal{O}(\mathcal{V})$ Iterationen.
- Wir hatten bereits festgestellt, dass jede Ausführung einer solchen Iteration $\mathcal{O}(\mathcal{E} \cdot \mathcal{V})$ Zeit kostet.
- Insgesamt ergibt sich also als Laufzeit: $\mathcal{O}(\mathcal{E} \cdot \mathcal{V}^2)$

  Laufzeit von Edmonds/Karp zum Vergleich: $\mathcal{O}(\mathcal{E}^2 \cdot \mathcal{V})$

Aber es geht noch besser!

Sei $G = (V, E, c, s, t)$ ein Netzwerk. Eine Funktion $f : E \to \mathbb{R}^+$ heißt *Präfluss*, falls gilt:

- $0 \leq f(u, v) \leq c(u, v)$ für alle $(u, v) \in E$
- Für alle Knoten $v \in V - \{s, t\}$ kann ein Überschuss $excess_f(v)$ vorhanden sein:

$$excess_f(v) := \sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e) \geq 0$$

Jeder Fluss f ist auch ein Präfluss, da $excess_f(v) = 0$ für alle Knoten $v \in V - \{s, t\}$ gilt. Im Folgenden sei $G_R(f) = (V, E_f, c_f)$ der Restgraph zu Graph G und Präfluss f.

Ein Knoten $v \in V - \{s, t\}$ heißt *aktiv* (active), wenn gilt: $excess_f(v) > 0$

Der Algorithmus von Goldberg und Tarjan wendet zwei Prozeduren PUSH und RELABEL an, um den Überschuss bei aktiven Knoten abzubauen.

Mittels $\textsc{Push}(u, v)$ wird ein Teil des Überschusses von Knoten u zu Knoten v verschoben.

Für die Ausführung müssen drei Bedingungen erfüllt sein:

- $\mathit{excess}_f(u) > 0$: Knoten u ist aktiv, d.h. an Knoten u liegt ein Überschuss vor.
- $(u, v) \in E_f$, also $c_f(u, v) > 0$: Im Restgraphen gibt es eine Kante von u nach v, über die der Überschuss weitergeleitet werden kann.
- $\mathit{height}(u) > \mathit{height}(v)$: Der Fluss kann nur von einem höheren zu einem niedrigeren Knoten fließen.

Der Wert, der verschoben wird, ist $\min\{\mathit{excess}_f(u), c_f(u, v)\}$.

Mittels $\textsc{Relabel}(u)$ kann der Knoten u angehoben werden, und zwar soweit, dass die Höhe um eins größer ist, als die Höhe des niedrigsten Nachbarknotens.

Für die Ausführung müssen zwei Bedingungen erfüllt sein:

- $\mathit{excess}_f(u) > 0$: Es muss einen Grund für die Erhöhung geben, d.h. es muss ein Überschuss abgetragen werden. Daher werden nur aktive Knoten angehoben.
- $\mathit{height}(u) \leq \mathit{height}(v)$ für alle Kanten $(u, v) \in E_f$: Alle zu u benachbarten Knoten liegen höher oder gleich hoch.

Durch das Anheben des Knotens u wird $\mathit{height}(u)$ auf den folgenden Wert gesetzt:

$$\mathit{height}(u) := \min\{\mathit{height}(v) + 1 \mid (u, v) \in E_f\}$$

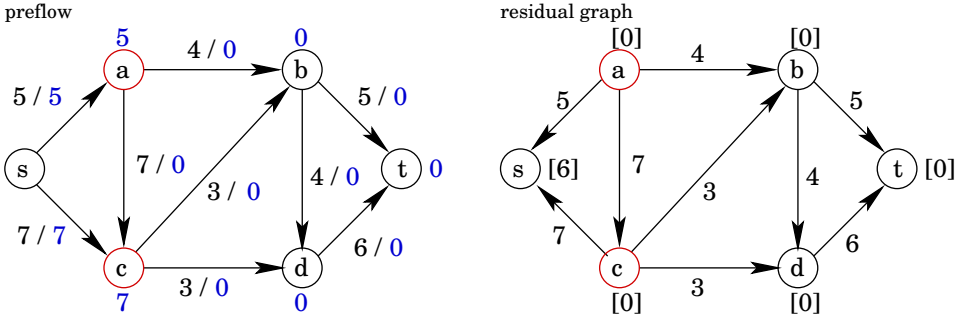
Eine Kante $(u, v) \in E_f$ heißt *zulässig* (admissible), wenn $\mathit{height}(u) = \mathit{height}(v) + 1$ gilt.

GoldbergTarjanAlgorithm()
  for all edges $(u, v) \in E$ **do** $f(u, v) := 0$
  for all neighbors v of s **do** $f(s, v) := c(s, v)$
  for all nodes $v \in V - \{s\}$ **do** $height(v) := 0$
  $height(s) := \mathcal{V}$

  while $\exists$ active node $v \in V - \{s, t\}$ **do**
    if $\exists$ admissible edge $(v, w) \in E_f$
      then Push(v,w)
      else Relabel(v)

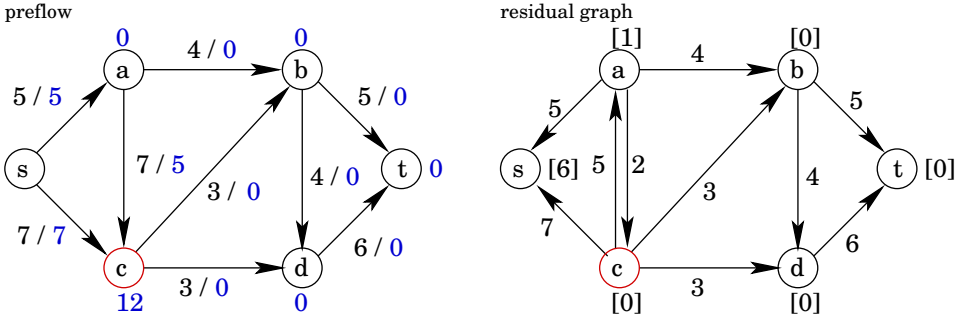
Push(v,w)
  if $(v, w) \in E$ **then**
    $f(v, w) := f(v, w) + \min\{excess_f(v), c_f(v, w)\}$
  else $f(v, w) := f(v, w) - \min\{excess_f(v), c_f(v, w)\}$

Relabel(v)
  $height(v) := \min\{height(w) + 1 \mid (v, w) \in E_f\}$



$\rightarrow$ RELABEL(a) und PUSH(a,c)

Damit der Überschuss in Richtung t abgebaut werden kann, werden Knoten immer nur soweit angehoben, dass der Überschuss in Richtung der kleinsten Höhe abfließen kann. Wir werden sehen, dass dadurch die Höhen der Knoten untere Schranken für die Länge der kürzesten Wege im Restgraphen zur Senke t sind.

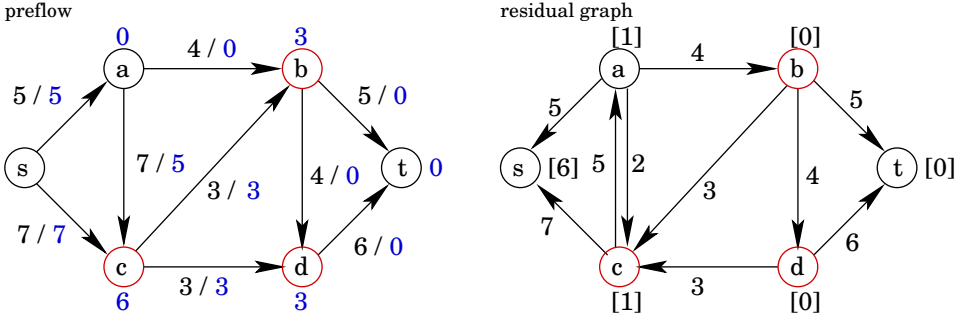


$\rightarrow$ RELABEL(c) und PUSH(c,b) und PUSH(c,d)

Invariante: Nach jeder PUSH- oder RELABEL-Operation gilt:

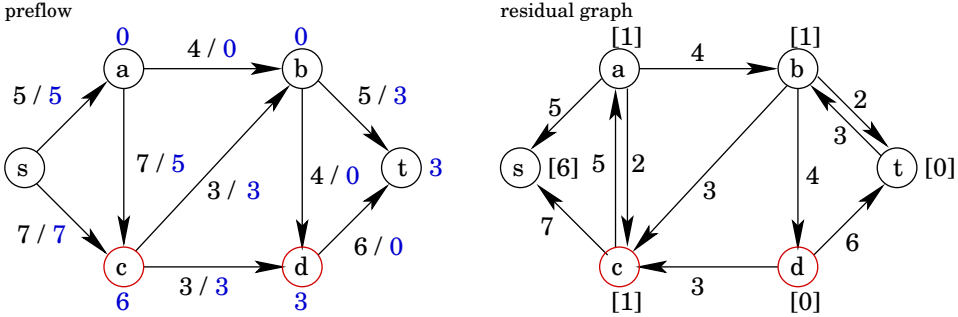
- f ist ein Präfluss.
- $height(u) \leq height(v) + 1$ für jede Kante $(u, v) \in E_f$ im Restgraphen.

Push-Relabel-Algorithmen



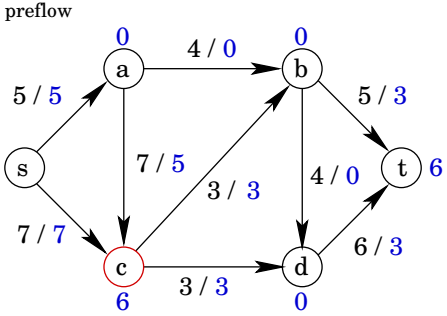
→ RELABEL(b) und PUSH(b,t)

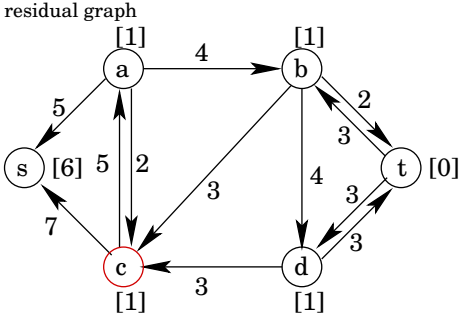
Push-Relabel-Algorithmen



→ RELABEL(d) und PUSH(d,t)

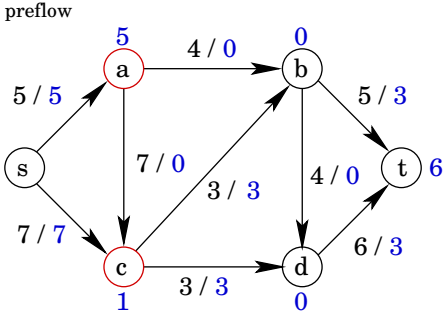
Push-Relabel-Algorithmen

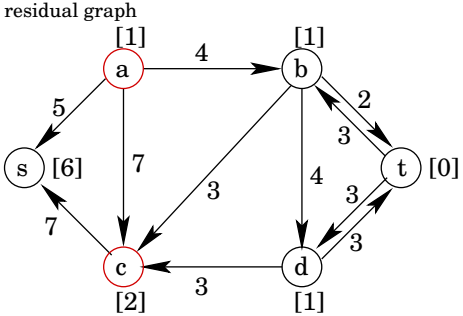




→ RELABEL(c) und PUSH(c,a)

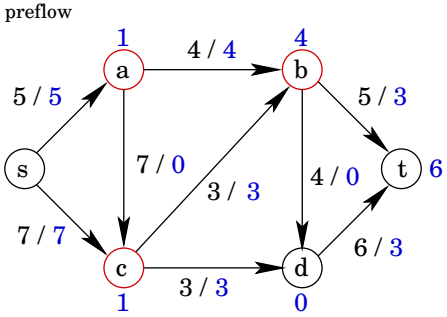
Push-Relabel-Algorithmen

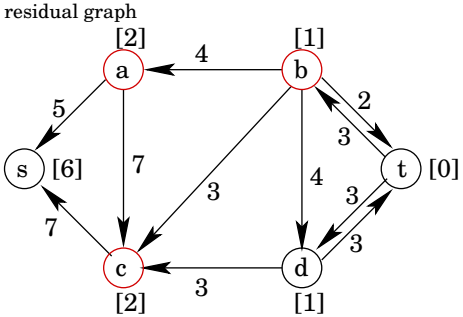




→ RELABEL(a) und PUSH(a,b)

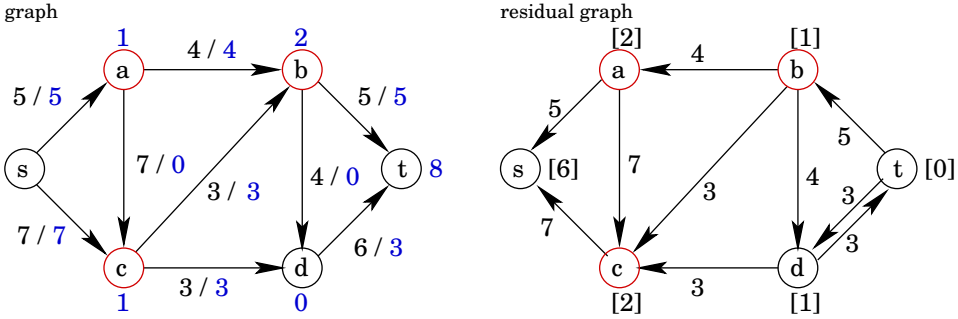
Push-Relabel-Algorithmen





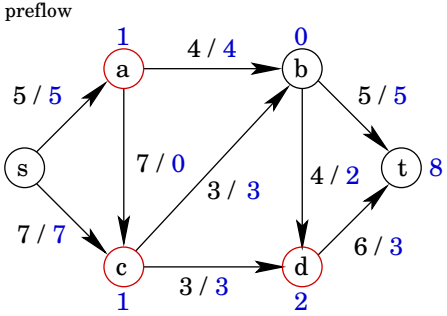
$\rightarrow$ PUSH(b,t)

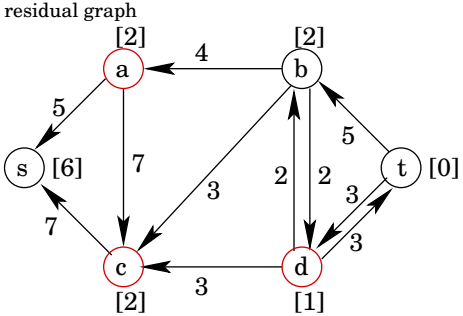
Push-Relabel-Algorithmen



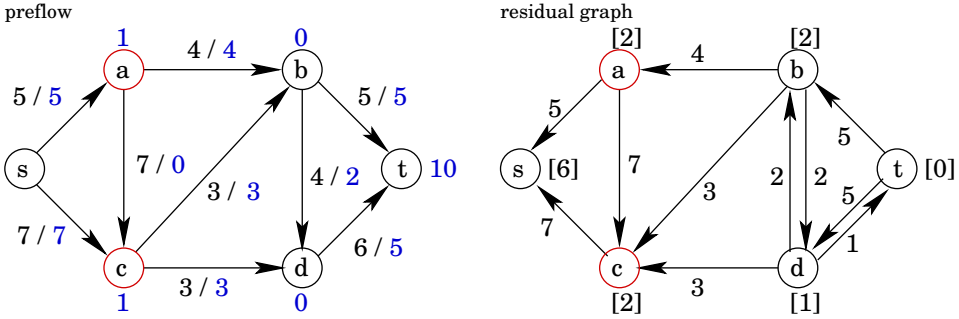
→ RELABEL(b) und PUSH(b,d)

Push-Relabel-Algorithmen





$\rightarrow$ PUSH(d,t)



Der Wert der Flussfunktion ist bereits maximal. Der Algorithmus muss aber noch die Überschüsse in den Knoten a und c abbauen. Dazu werden a und c immer weiter angehoben, bis der Überschuss in Richtung s abgebaut werden kann.

Invariante:

Nach jeder PUSH- oder RELABEL-Operation gilt: f ist ein Präfluss.

RELABEL:

- Der Präfluss wird nicht verändert.

PUSH(u, v):

- Da niemals mehr als der Überschuss $excess_f(u)$ in Knoten u abgebaut wird, bleibt $excess_f(u) \geq 0$ auch nach der Operation erhalten.
- Da der Präfluss an Knoten v erhöht wird, und bereits vor dem Erhöhen $excess_f(v) \geq 0$ galt, gilt dies auch nach dem PUSH.

Invariante:

Nach jeder PUSH- oder RELABEL-Operation gilt für jede Kante $(u, v) \in E_f$ im Restgraphen: $height(u) \leq height(v) + 1$

RELABEL(u):

- Vor dem Erhöhen galt $height(u) \leq height(v) + 1$ für alle Kanten $(u, v) \in E_f$.
- Durch das Erhöhen wird $height(u)$ auf den Wert $\min\{height(v) + 1 \mid (u, v) \in E_f\}$ gesetzt.

PUSH(u, v):

- Es kann eine neue Kante (v, u) im Restgraphen entstehen.
- Vor der Ausführung von PUSH galt $height(u) = height(v) + 1$, da nur zulässige Kanten ausgewählt werden.

$\rightarrow$ Daher gilt nachher: $height(v) = height(u) - 1 \leq height(u) + 1$

Senke t ist im Restgraphen von Quelle s aus nicht erreichbar. Beweis durch Widerspruch:

- Annahme: Es existiert ein einfacher gerichteter Weg von s nach t im Restgraphen $G_R(f)$.
- Sei $s = v_0 \to v_1 \to v_2 \to \cdots \to v_l = t$ ein solcher Weg.
- Dann gilt $height(v_i) \leq height(v_{i+1}) + 1$ für $0 \leq i < l$.
- Also gilt $height(s) \leq height(t) + l$ und $l < \mathcal{V}$.
- ↯ Initial wird $height(s) = \mathcal{V}$ und $height(t) = 0$ gesetzt, und t wird niemals angehoben.

Der Algorithmus terminiert, wenn es keine aktiven Knoten gibt:

- Dann gilt $excess_f(u) = 0$ für alle Knoten $u \in V - \{s, t\}$, und daher ist f ein Fluss.
- Außerdem gibt es einen Schnitt S, so dass $s \in S$ und $t \in \overline{S}$ ist.

→ Also ist f maximal.

Wir müssen noch zeigen, dass der Algorithmus terminiert.

Sei f ein Präfluss. Wenn $excess_f(u) > 0$ für einen Knoten u gilt, dann ist Quelle s im Restgraphen von u aus erreichbar, d.h. der Überschuss kann in Richtung Quelle abgebaut werden.

- Sei U die Menge aller von u aus erreichbaren Knoten im Restgraphen. Zu zeigen: $s \in U$
- Nach Definition von $excess_f(v)$ gilt:

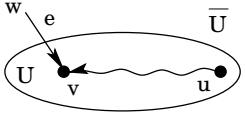
  $$\sum_{v \in U} excess_f(v) = \sum_{v \in U} \left(\sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e) \right)$$

  Dabei gilt:
  - $e \in U \times U$ kommt positiv und negativ vor: Beitrag 0
  - $e \in \overline{U} \times \overline{U}$ kommt nicht vor
  - $e \in \overline{U} \times U$ kommt nur positiv vor: Beitrag $f(e)$
  - $e \in U \times \overline{U}$ kommt nur negativ vor: Beitrag $-f(e)$

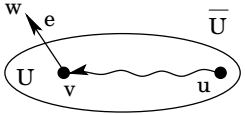
$$\rightarrow \sum_{v \in U} excess_f(v) = \sum_{e \in E \cap (\overline{U} \times U)} f(e) - \sum_{e \in E \cap (U \times \overline{U})} f(e)$$

Auf der letzten Folie hatten wir festgestellt:

$$\sum_{v \in U} excess_f(v) = \sum_{e \in E \cap (\overline{U} \times U)} f(e) - \sum_{e \in E \cap (U \times \overline{U})} f(e)$$



Es muss $f(w, v) = 0$ gelten, da sonst die Kante $(v, w) \in E_f$ erzeugt würde, und damit $w \in U$ gelten würde.



Es muss $f(v, w) = c(e)$ gelten, damit die Kante (v, w) nicht im Restgraphen liegt, denn sonst würde $w \in U$ gelten.

$$\sum_{v \in U} excess_f(v) = \sum_{e \in E \cap (\overline{U} \times U)} 0 - \sum_{e \in E \cap (U \times \overline{U})} c(e) = - \sum_{e \in E \cap (U \times \overline{U})} c(e)$$

Auf der letzten Folie hatten wir festgestellt:

$$\sum_{v \in U} excess_f(v) = - \sum_{e \in E \cap (U \times \overline{U})} c(e)$$

Annahme: $s \notin U$

Da $excess_f(v) \geq 0$ für alle Knoten $v \in V - \{s, t\}$ gilt, muss gelten:

$$\sum_{v \in U} excess_f(v) = - \sum_{e \in E \cap (U \times \overline{U})} c(e) \geq 0$$

Da $c(e) \geq 0$ für alle Kanten $e \in E$ gilt, kann nur gelten:

$$\sum_{v \in U} excess_f(v) = - \sum_{e \in E \cap (U \times \overline{U})} c(e) = 0 \; \lightning$$

Widerspruch, da $excess_f(u) > 0$ ist und $u \in U$ gilt.

Ein Überschuss kann also zur Quelle s abgebaut werden. Nun zeigen wir, dass die Knoten nicht beliebig angehoben werden.

Es gilt: $\mathit{height}(v) \leq 2\mathcal{V} - 1$ für alle $v \in V$.

- Sei v ein aktiver Knoten.
- Dann gibt es einen Weg $v = v_0 \to v_1 \to v_2 \to \cdots \to v_l = s$ der Länge $l \leq \mathcal{V} - 1$.
- Außerdem gilt $\mathit{height}(v_i) \leq \mathit{height}(v_{i+1}) + 1$ für $0 \leq i < l$.
- Da initial $\mathit{height}(s) = \mathcal{V}$ gesetzt wird und Knoten s nicht angehoben wird, gilt: $\mathit{height}(v) \leq \mathit{height}(s) + \mathcal{V} - 1 = 2\mathcal{V} - 1$

Es werden höchstens $\mathcal{O}(\mathcal{V}^2)$ RELABEL-Operationen durchgeführt, denn jeder der $\mathcal{V}$ Knoten kann höchstens $2\mathcal{V} - 1$ mal angehoben werden.

Operation $\textsc{Push}(u, v)$ heißt *sättigend*, falls $c_f(u, v) \leq \mathit{excess}_f(u)$ gilt, ansonsten heißt die Operation *nicht sättigend*.

Es werden $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ sättigende $\textsc{Push}$-Operationen durchgeführt:

- Wir betrachten eine beliebige Kante $e = (u, v)$.
- Wird Kante e saturiert, dann gilt $\mathit{height}(u) > \mathit{height}(v)$.
- Kante e wird aus dem Restgraphen entfernt und ist erst dann wieder enthalten, wenn ein $\textsc{Push}(v, u)$ ausgeführt wurde.
- Dazu muss $\mathit{height}(v) > \mathit{height}(u)$ gelten, also muss $\mathit{height}(v)$ um mindestens zwei erhöht worden sein.
- Damit also ein weiteres Mal ein $\textsc{Push}(u, v)$ Kante e sättigen kann, muss auch $\mathit{height}(u)$ um mindestens zwei steigen.
- $\mathit{height}(u)$ kann höchstens auf $2\mathcal{V} - 1$ steigen, also kann Kante e höchstens $\frac{2\mathcal{V}-1}{2} < \mathcal{V}$ mal saturiert werden.
- Da es im Restgraphen höchstens $2 \cdot \mathcal{E}$ viele Kanten gibt, kann es nur $2\mathcal{E} \cdot \mathcal{V}$ sättigende $\textsc{Push}$-Operationen geben.

Um zu zeigen, dass $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$ nicht sättigende PUSH-Operationen durchgeführt werden, benötigen wir folgende Potenzialfunktion:

$$\Phi(f) := \sum_{v : excess_f(v) > 0} height(v)$$

nicht sättigende $\text{PUSH}(v, w)$-Operation:

- Der Wert der Potenzialfunktion wird um $height(v)$ verringert, da der Überschuss an Knoten v komplett abgebaut wird.
- Allerdings wird der Wert ggf. um $height(w)$ erhöht, da der Überschuss nun bei Knoten w ist, und Knoten w vorher evtl. keinen Überschuss hatte.
- Da aber $height(v) > height(w)$ sein muss, wird der Wert insgesamt um mindestens eins erniedrigt.

sättigende $\textsc{Push}(v, w)$-Operation:

- Wir wissen bereits, dass $\mathit{height}(w) \leq 2\mathcal{V} - 1$ ist, also wird der Wert der Potenzialfunktion um höchstens $2\mathcal{V} - 1$ erhöht.
- Da es nur $2\mathcal{V} \cdot \mathcal{E}$ sättigende $\textsc{Push}$-Operationen gibt, kann der Potenzialanstieg nach oben mit $4\mathcal{V}^2 \cdot \mathcal{E}$ abgeschätzt werden.

$\textsc{Relabel}(v)$:

- Da der Knoten v angehoben wird, steigt auch der Wert der Potenzialfunktion.
- Insgesamt ist der Potenzialanstieg durch $(2\mathcal{V} - 1) \cdot \mathcal{V} \in \mathcal{O}(\mathcal{V}^2)$ beschränkt, da $\mathit{height}(v) \leq 2\mathcal{V} - 1$ ist und es $\mathcal{V}$ viele Knoten gibt.

Wir können also festhalten:

- Der Potenzialanstieg durch sättigende PUSH-Operationen kann nach oben mit $4\mathcal{V}^2 \cdot \mathcal{E}$ abgeschätzt werden.
- Der Potenzialanstieg durch RELABEL-Operationen ist durch $2\mathcal{V}^2$ nach oben beschränkt.
- Bei jeder nicht sättigenden PUSH-Operation wird der Wert der Potenzialfunktion um mindestens eins erniedrigt.

Für die Anzahl der nicht sättigenden PUSH-Operationen np gilt:

$$np \leq 2\mathcal{V}^2 + 4\mathcal{V}^2 \cdot \mathcal{E} \in \mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$$

Die Laufzeit des Goldberg-Tarjan-Algorithmus ist also $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$:

- $\mathcal{O}(\mathcal{V}^2)$ RELABEL-Operationen
- $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$ sättigende PUSH-Operationen
- $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$ nicht sättigende PUSH-Operationen

Wir haben noch nicht festgelegt, wie der aktive Knoten gewählt wird. Diese Wahl hat entscheidenden Einfluss auf die Laufzeit:

- Wähle von allen aktiven Knoten denjenigen Knoten aus, dessen *height*-Wert am größten ist. $\rightarrow$ Laufzeit: $\mathcal{O}(\mathcal{V}^2 \cdot \sqrt{\mathcal{E}})$

  Wir zeigen nur: Laufzeit $\mathcal{O}(\mathcal{V}^3)$
  - Ein Knoten v wird solange gewählt, bis er durch eine nicht sättigende PUSH-Operation nicht mehr aktiv ist.
  - Knoten v kann erst wieder aktiv werden, wenn für mindestens einen Knoten w der Wert $\mathit{height}(w)$ erhöht wurde.
  - Nach $\mathcal{V}$ vielen nicht sättigenden PUSH-Operationen ohne zwischenzeitlicher RELABEL-Operation gibt es keine aktiven Knoten mehr und der Algorithmus terminiert.
  - Da es nur $\mathcal{O}(\mathcal{V}^2)$ viele RELABEL-Operationen gibt, kann es nur $\mathcal{O}(\mathcal{V}^3)$ viele nicht sättigende PUSH-Operationen geben.
- Verwalte die aktiven Knoten in einer FIFO-Liste.
  $\rightarrow$ Laufzeit: $\mathcal{O}(\mathcal{V}^3)$

Verallgemeinerung vom Max-Flow-Problem

Wir führen für die Kanten auch minimale Kapazitäten ein.

Gegeben:
- Ein gewichteter Graph $G = (V, E, \ell, u)$ ohne antiparallele Kanten mit zwei Kostenfunktionen $\ell, u : E \to \mathbb{Q}_0^+$.
- Eine Quelle $s \in V$ und eine Senke $t \in V$, wobei $s \neq t$ gilt.

Gesucht: Maximaler Fluss im Netzwerk von Quelle s nach Senke t, falls ein zulässiger Fluss überhaupt existiert.

Eine *Flussfunktion* $f : E \to \mathbb{Q}_0^+$ muss nun auch die minimale Kapazität berücksichtigen.

- Kapazitätsbeschränkung: $\forall e \in E : \ \ell(e) \leq f(e) \leq u(e)$
- Flusserhaltung: $\forall v \in V - \{s, t\} :$

$$\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$$

Der initiale Fluss $f(e) = 0$ für alle Kanten $e \in E$ ist ggf. nicht mehr zulässig.

Verallgemeinerung vom Max-Flow-Problem

Lösung des Max-Flow-Problems in zwei Phasen:

1. Finde einen zulässigen, initialen Fluss.
2. Verbessere die Lösung mittels bekannter Flussalgorithmen.

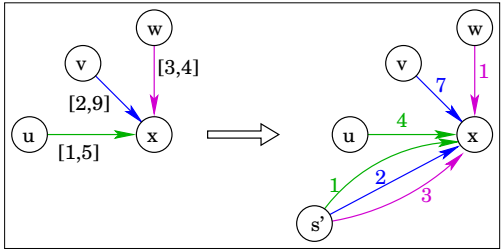
Um einen initialen, zulässigen Fluss zu bestimmen, lösen wir das Max-Flow-Problem auf einem modifizierten Graphen $G' = (V', E', c)$:

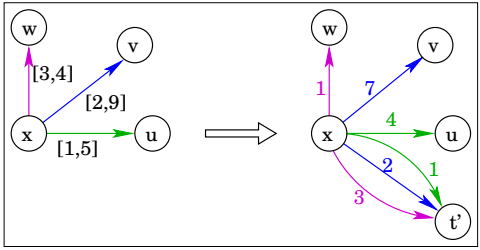
- Füge eine neue Kante (t, s) mit Kapazität $c((t, s)) = \infty$ hinzu.
- Füge eine neue Quelle s' und eine neue Senke t' hinzu: $V' := V \cup \{s', t'\}$
- Füge Kanten (s', v) mit $c((s', v)) := \sum_{(u,v)\in E} \ell((u, v))$ für $v \in V$ hinzu.
- Füge Kanten (v, t') mit $c((v, t')) := \sum_{(v,w)\in E} \ell((v, w))$ für $v \in V$ hinzu.
- Für jede Kante $e \in E$ setze $c(e) := u(e) - \ell(e)$.

Im Prinzip ersetzen wir jede Kante $(a, b) \in E$ durch drei Kanten in E':

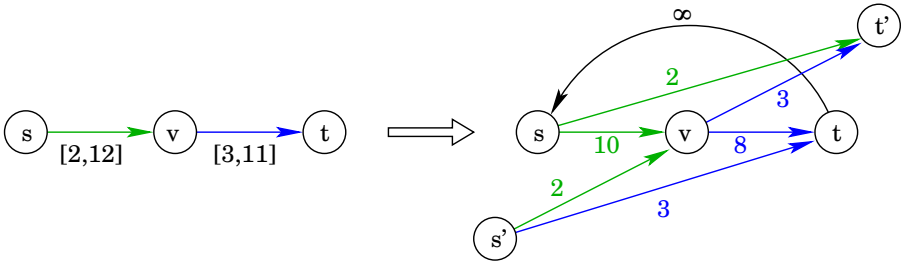
- (s', b) mit $c((s', b)) := \ell((a, b))$
- (a, t') mit $c((a, t')) := \ell((a, b))$
- (a, b) mit $c((a, b)) := u((a, b)) - \ell((a, b))$

Durch die ersten beiden Schritte können parallele Kanten entstehen: Fasse parallele Kanten zu einer einzigen Kante zusammen; die Kapazität der Kante ergibt sich aus der Summe der Kapazitäten der parallelen Kanten.



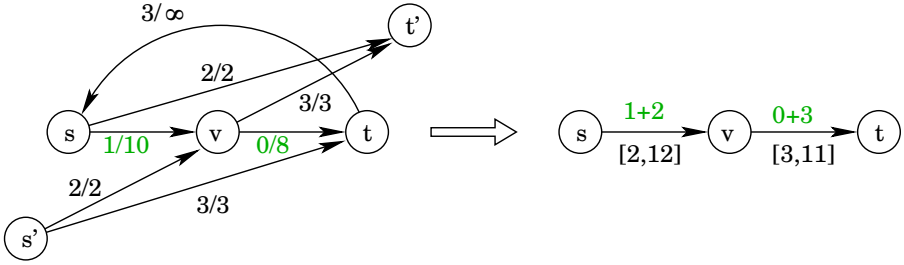


Beispiel: Zunächst transformieren wir das gegebene Netzwerk G in das Netzwerk G'.



Auf dem Netzwerk G' lösen wir dann das Max-Flow-Problem, um einen initialen Fluss für G zu bestimmen.

Nachdem der maximale Fluss f' im modifizierten Netzwerk G' bestimmt wurde (links im Bild), wird dieser Fluss zu einem zulässigen Fluss in dem originalen Netzwerk G transformiert (rechts): Addiere Fluss $f'(e)$ auf untere Grenze $\ell(e)$ der Kante e auf.



Beobachtung: Obwohl wir einen maximalen Fluss im modifizierten Netzwerk berechnet haben, erhalten wir einen minimalen Fluss im originalen Netzwerk!

Noch zu zeigen: Ein maximaler Fluss im modifizierten Netzwerk G' entspricht einem zulässigen Fluss im originalen Netzwerk G. Zunächst stellen wir fest:

$$\begin{aligned} L &:= \sum_{v \in V} c((s', v)) \overset{Def.}{=} \sum_{v \in V} \sum_{(u,v) \in E} \ell((u, v)) \\ &= \sum_{v \in V} c((v, t')) \overset{Def.}{=} \sum_{v \in V} \sum_{(v,w) \in E} \ell((v, w)) = \sum_{(a,b) \in E} \ell((a, b)) \end{aligned}$$

Denn der Wert $\ell((a, b))$ für eine beliebige Kante $(a, b) \in E$ ist in $\sum_{v \in V} c((s', v))$ enthalten für $v = b$ und in $\sum_{v \in V} c((v, t'))$ enthalten für $v = a$.

Wir nennen einen Fluss in G' *saturiert*, wenn jede Kapazität $c((s', v))$ und $c((v, t'))$ voll ausgeschöpft wird, wenn also $f'((s', v)) = c((s', v))$ und $f'((v, t')) = c((v, t'))$ für alle $v \in V$ gilt und daher $F_{f'} = L$ ist.

Lemma: G hat genau dann einen zulässigen (s, t)-Fluss, wenn G' einen saturierten (s', t')-Fluss hat.

Beweis: "⇒" Sei $f : E \to \mathbb{Q}^+$ ein zulässiger (s,t)-Fluss in G. Dann gilt

- $\ell(e) \leq f(e) \leq u(e)$ für alle Kanten $e \in E$ und
- $\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$ für alle Knoten $v \in V - \{s,t\}$.

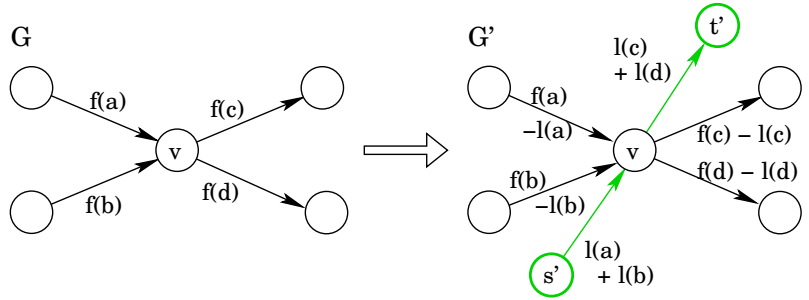
Betrachten wir dazu die folgende Funktion $f' : E' \to \mathbb{Q}^+$ in G':

- $f'((s',v)) := \sum_{(u,v) \in E} \ell((u,v))$
- $f'((v,t')) := \sum_{(v,w) \in E} \ell((v,w))$
- $f'((u,v)) := f((u,v)) - \ell((u,v))$
- $f'((t,s)) := F_f$

Dass f' saturiert ist, folgt unmittelbar aus den beiden linken Festlegungen. Es bleibt aber zu zeigen, dass f' ein zulässiger (s',t')-Fluss in G' ist.

Es gilt $0 \leq f'(e) = f(e) - \ell(e) \leq u(e) - \ell(e) = c(e)$, weil $\ell(e) \leq f(e) \leq u(e)$ für alle Kanten $e \in E$ gilt, denn f ist ein zulässiger Fluss in G.

Wegen $\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$ gilt die Flusserhaltung auch für f' in jedem Knoten.

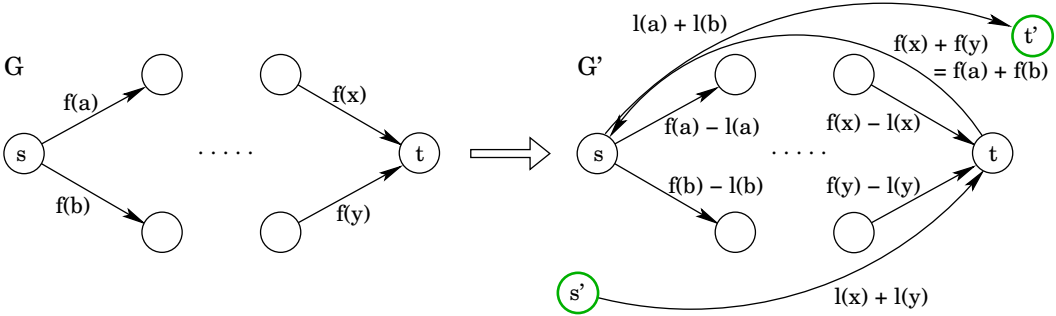


Denn es gilt:

$$\sum_{e \in In(v)} f(e) - \ell(e) + \sum_{(u,v) \in E} \ell((u,v)) = \sum_{e \in Out(v)} f(e) - \ell(e) + \sum_{(v,w) \in E} \ell((v,w))$$

$$\sum_{e \in In(v)} f'(e) + f'((s',v)) = \sum_{e \in Out(v)} f'(e) + f'((v,t'))$$

Auch für die Knoten s und t gilt die Flusserhaltung für f' in G', weil die Kante (t, s) hinzugefügt wurde.

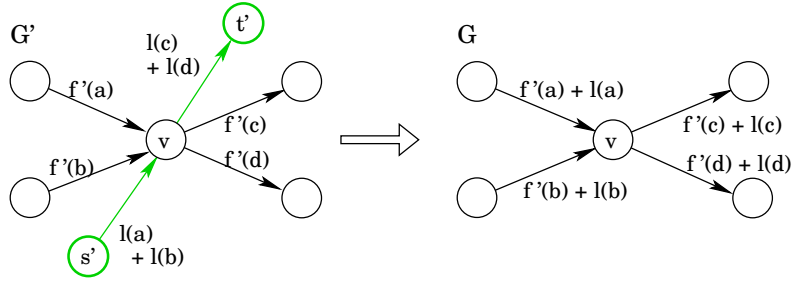


Beweis: "$\Leftarrow$" Sei f' ein saturierter Fluss in G'. Dann sei $f(e) := f'(e) + \ell(e)$ für alle Kanten $e \in E$ definiert. Es gilt:

Die Kapazitätsbegrenzung $\ell(e) \leq f(e) \leq u(e)$ für G wird eingehalten:

$$0 \leq f'(e) = f(e) - \ell(e) \leq u(e) - \ell(e) = c(e)$$

Die Flusserhaltung in G ist erfüllt, weil f' saturiert ist:

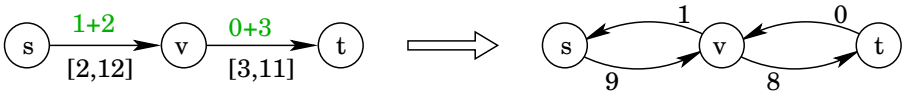


Sobald ein zulässiger, initialer Fluss für G berechnet wurde, kann dieser mit den bekannten Algorithmen verbessert werden.

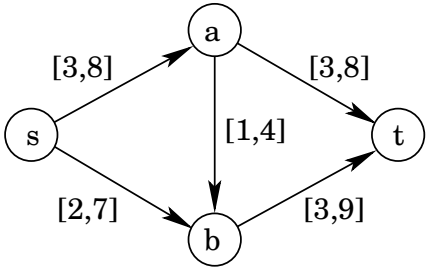
Um dabei sicherzustellen, dass jeder Fluss zulässig ist, muss die Restkapazität einer Kante wie folgt definiert werden:

$$c_f((a,b)) = \begin{cases} u((a,b)) - f((a,b)) & \text{falls } (a,b) \in E \\ f((b,a)) - \ell((b,a)) & \text{falls } (b,a) \in E \\ 0 & \text{sonst} \end{cases}$$

im Beispiel:



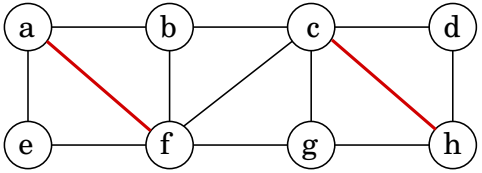
Übung 34. *Bestimmen Sie zu dem folgenden Netzwerk mit oberen und unteren Schranken einen initialen, zulässigen Fluss.*



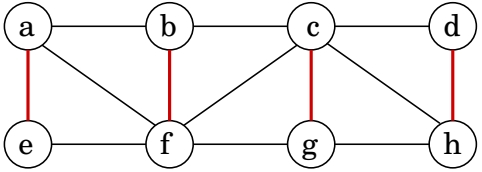
Übersicht

Sei $G = (V, E)$ ein ungerichteter Graph. Ein *Matching* ist eine Teilmenge $M \subseteq E$ der Kanten, so dass keine zwei Kanten aus M einen gemeinsamen Endknoten haben:

- *Maximales Matching* $\rightarrow$ nicht vergrößerbares Matching



- *Maximum-Matching* $\rightarrow$ ein größtes Matching

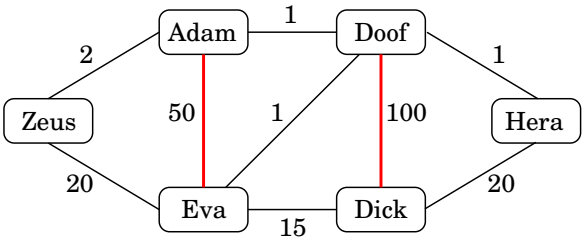


[12] Paarung, Abgleich, Anpassung

Für gewichtete Graphen $G = (V, E, c)$ ist $M \subseteq E$ ein *maximum weight matching*, wenn für alle Matchings $M' \subseteq E$ gilt:

$$\sum_{e \in M} c(e) \geq \sum_{e \in M'} c(e)$$

Beispiel: [13]



Wir werden uns auf die Berechnung von größten (maximum) Matchings beschränken.

[13] Quelle Bild: Ottmann, Widmayer: Algorithmen und Datenstrukturen.

Matching-Probleme sind in vielen Bereichen zu lösen:

- Stundenplanerstellung: Finde eine Zuordnung zwischen Raum, Fach und Lehrer.
  $\rightarrow$ dreidimensionales Matching, NP-vollständig
- Sind die Zuordnungen auf disjunkten Knotenmengen zu berechnen, also z.B. wenn
  - Studenten auf Studienplätze verteilt werden sollen, oder
  - den Mitarbeitern Tätigkeiten zugewiesen werden sollen,

  so spricht man von bipartiten Matchings.

Ein ungerichteter Graph $G = (V, E)$ heißt *bipartit*, wenn sich die Knotenmenge in 2 Mengen $V_1, V_2 \subseteq V$ zerlegen lässt, so dass gilt:

- $V_1 \cup V_2 = V$ und $V_1 \cap V_2 = \emptyset$
- Für alle $\{u, v\} \in E$ gilt: $u \in V_1,\ v \in V_2$ oder $u \in V_2,\ v \in V_1$

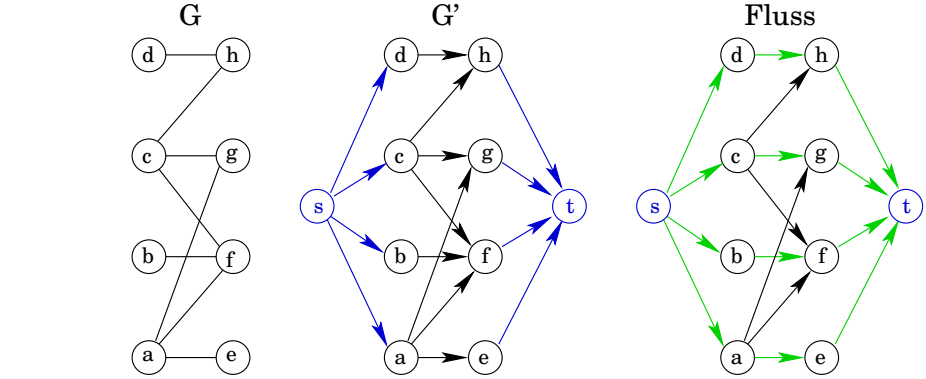
zunächst: Berechne ein größtes Matching für bipartite Graphen mittels Flussalgorithmus.

Konstruiere zu dem gegebenen bipartiten Graphen $G = (V, E)$ mit $V = V_1 \cup V_2$ das Netzwerk $G' = (V', E', c)$ mit

- $V' = V \cup \{s, t\}$
- $E' = \{s\} \times V_1 \ \cup \ \{(u, v) \in V_1 \times V_2 \mid \{u, v\} \in E\} \ \cup \ V_2 \times \{t\}$
- $c : E' \to \mathbb{N}$ mit $c(e) = 1$ für alle Kanten $e \in E'$.

Algorithmus

- Bestimme für G' einen maximalen Fluss durch zunehmende Pfade P mit $\Delta(P) = 1$.
  $\rightarrow$ Jede Kante in G' hat den Flusswert 0 oder 1.
- Die Kanten $\{u, v\} \in E$ mit $f((u, v)) = 1$ bilden ein größtes Matching in G.



größtes Matching: Alle Kanten $\{u, v\} \in E$ mit $f((u, v)) = 1$

- Für alle $v \in V_1$ gilt: Falls $f(s, v) = 1$ ist, dann hat genau eine auslaufende Kante $e = (v, w)$ den Fluss $f(e) = 1$.
- Für alle $w \in V_2$ gilt: Falls $f(w, t) = 1$ ist, dann hat genau eine einlaufende Kante $e = (v, w)$ den Fluss $f(e) = 1$.
- Also gilt: $f(s, v) = 1 \Rightarrow f(v, w) = 1 \Rightarrow f(w, t) = 1$ liefert Matching-Kante (v, w).

Es geht auch ohne Flussalgorithmen!

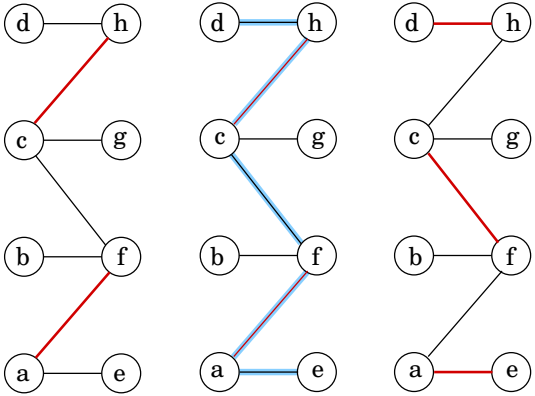
Sei $G = (V, E)$ ein Graph, und sei $M \subseteq E$ ein Matching.

- Die Kanten aus M heißen *gebundene Kanten*, die übrigen sind die *freien Kanten*.
- Die Endknoten der gebundenen Kanten heißen *gebundene Knoten*, alle übrigen Knoten sind *freie Knoten*.
- Ein Weg $P = (v_1, \ldots, v_k)$, der abwechselnd aus freien und gebundenen Kanten besteht, heißt *alternierender Weg*.
- Ein alternierender Weg $P = (v_1, \ldots, v_k)$ heißt *zunehmend alternierender Weg*, wenn $v_1 \neq v_k$ gilt und beide Endknoten v_1 und v_k mit keiner gebundenen Kante inzident sind.

  Achtung: Nach dieser Definition ist auch eine einzelne, freie Kante ein zunehmend alternierender Weg. Wir können eine solche Kante dem Matching hinzufügen.

Wir nutzen zunehmend alternierende Wege, um ein bestehendes Matching zu vergrößern. Beachte: $|M| \leq \mathcal{V}/2$

Matching in bipartiten Graphen



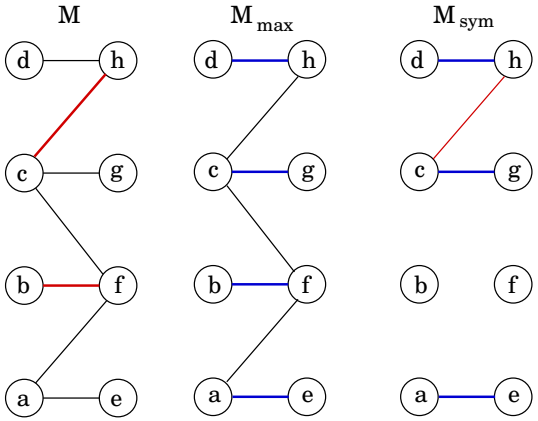
Das bestehende Matching M kann vergrößert werden, indem

- ein zunehmend alternierender Weg P gesucht wird,
- und dann alle gebundenen Kanten in P aus M entfernt werden und alle freien Kanten in P zu M hinzugefügt werden.

Sei M ein beliebiges Matching, und sei M_{max} ein größtes Matching. Die *symmetrische Differenz*

$$M_{sym} = (M_{max} - M) \cup (M - M_{max})$$

enthält genau die Kanten, die nur in einem der beiden Matchings vorkommen.



Durch fortlaufendes Finden zunehmend alternierender Wege kann ein größtes Matching berechnet werden:

- Für $M_{sym} = (M_{max} - M) \cup (M - M_{max})$ gilt: Jeder Knoten im Graphen G ist mit höchstens zwei Kanten aus M_{sym} inzident: eine Kante aus M_{max}, und eine Kante aus M
- Die Anzahl der aus M_{max} stammenden Kanten ist größer als die Anzahl der aus M stammenden Kanten: $|M_{max}| > |M|$ und in M_{sym} sind alle Kanten aus $M_{max} \cup M$, die nicht in beiden Mengen sind.
- $G_{sym} = (V, M_{sym})$ besteht aus knotendisjunkten Wegen bzw. Kreisen, die abwechselnd Kanten aus M_{max} und M benutzen.
- Jeder Kreis, falls einer existiert, hat genau so viele Kanten aus M_{max} wie aus M.

$\rightarrow$ Da $|M_{max}| > |M|$ ist, muss es Wege geben, die eine Kante mehr aus M_{max} als aus M haben, und dies sind zunehmend alternierende Wege.

Wir stellen fest:

- Wenn ein zunehmend alternierender Weg existiert, kann das bisherige Matching vergrößert werden.
- Falls kein solcher Weg gefunden wird, terminiert der Algorithmus und hat ein größtes Matching berechnet.

In bipartiten Graphen können zunehmend alternierende Wege mittels Breitensuche in Zeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$ berechnet werden. Daher ergibt sich für obigen Algorithmus die Laufzeit:

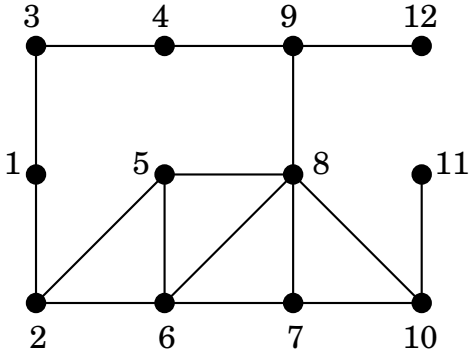
- maximal $\mathcal{V}/2$ Aufrufe der Breitensuche

$\rightarrow$ insgesamt: $\mathcal{O}(\mathcal{V}^2 + \mathcal{V} \cdot \mathcal{E})$ oder $\mathcal{O}(\mathcal{V}^3)$ wegen $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$

Der Algorithmus von Hopcroft und Karp findet in jedem Durchlauf alle kürzesten, zunehmend alternierenden Wege und benötigt daher nur $\sqrt{\mathcal{V}}$ Durchläufe.

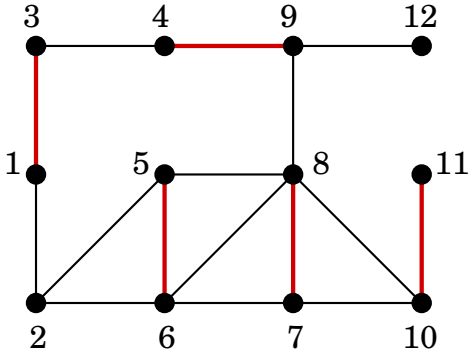
$\rightarrow$ Laufzeit $\mathcal{O}(\sqrt{\mathcal{V}} \cdot \mathcal{E})$

Matching auf allgemeinen Graphen



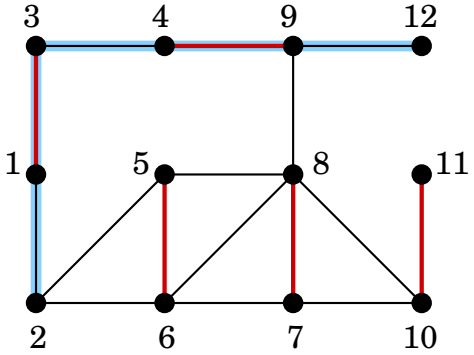
$\rightarrow$ bestimme ein initiales Matching

Matching auf allgemeinen Graphen

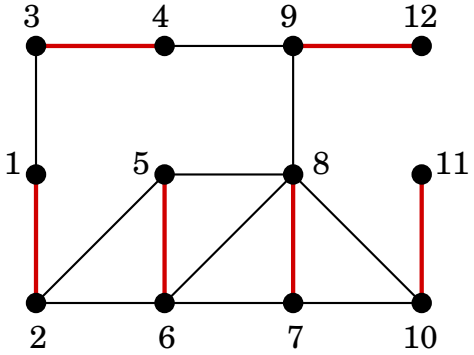


$\rightarrow$ bestimme einen zunehmend alternierenden Weg

Matching auf allgemeinen Graphen

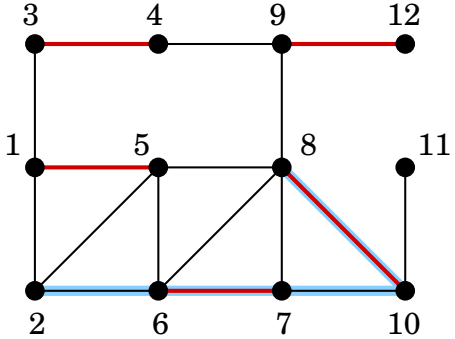


$\rightarrow$ vergrößere das Matching entlang dieses Weges

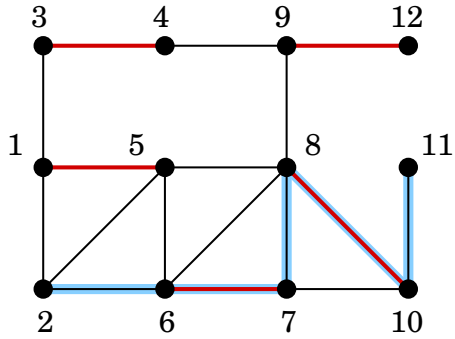


Leider hilft uns die Breitensuche allein nun nicht weiter. Es kann vorkommen, dass ein Knoten auf einem Weg gerader Länge und auf einem anderen Weg ungerader Länge erreichbar ist.

Starte Breitensuche bei einem freien Knoten: 2



Der Weg $P_1 = (2, 6, 7, 10, 8)$ hat die Länge 4, also gerade Länge.

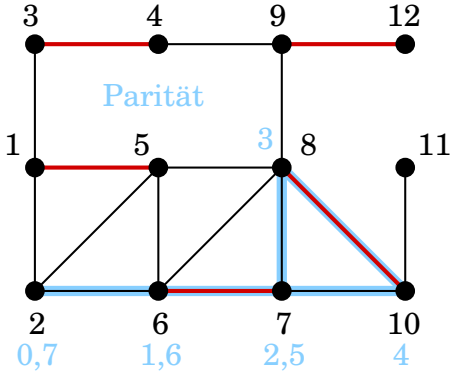


Der Weg $P_2 = (2, 6, 7, 8)$ hat die Länge 3, also ungerade Länge.

Der Weg P_1 kann nicht zu einem zunehmend alternierenden Weg erweitert werden! Würde die Breitensuche bei Knoten 7 anders verzweigen, würde ein solcher Weg gefunden.

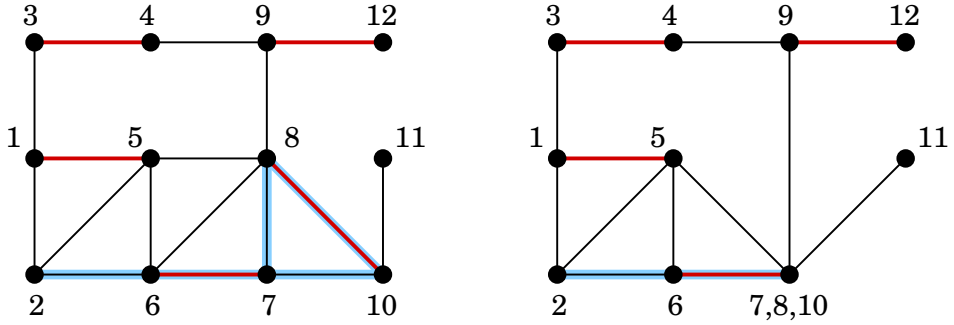
Idee: Die Breitensuche darf jeden Knoten zweimal besuchen, einmal für gerade Entfernung und einmal für ungerade Entfernung.

Das ist leider keine gute Idee: Es kann dann passieren, dass wir einen Kreis ungerader Länge komplett durchlaufen, und anschließend einen Teil des Weges rückwärts gehen, da sich die Paritäten der Knoten umgedreht haben. Ein solcher Weg ist natürlich nicht zunehmend alternierend.



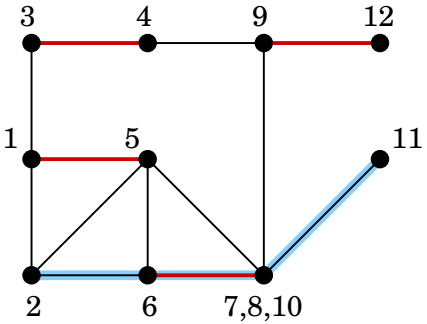
Der Weg $P_3 = (2, 6, 7, 8, 10, 7, 6, 2)$ läuft einmal komplett durch den Kreis $(7, 8, 10, 7)$ und läuft dann zum Startknoten der Breitensuche zurück. Dies ist möglich, da alle Knoten einmal für gerade und einmal für ungerade Entfernung besucht werden dürfen.

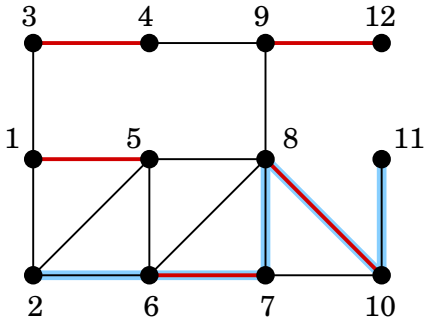
Ein Kreis ungerader Länge heißt *Blüte*. Wird bei der Suche nach einem zunehmend alternierenden Weg eine Blüte gefunden, so schrumpfen wir die Blüte auf einen Knoten zusammen: Jede Kante, die zuvor mit einem Knoten der Blüte verbunden war, führt jetzt zu diesem neuen Knoten.



Der Weg kann zu einem zunehmend alternierenden Weg erweitert werden.

Nachdem ein zunehmend alternierender Weg gefunden wurde, müssen alle geschrumpften Blüten wieder expandiert werden.

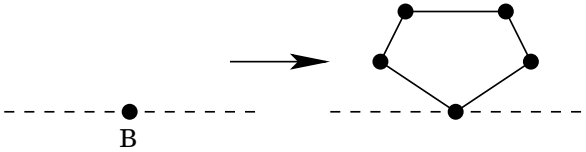




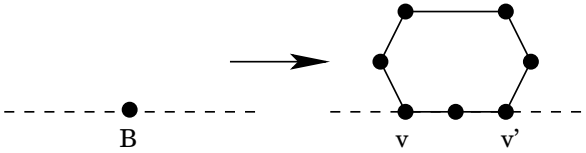
Dazu muss nur bestimmt werden, wie der zunehmend alternierende Weg innerhalb der expandierten Blüte fortgesetzt wird. Das dies immer möglich ist, zeigt die folgende Überlegung.

Sei P ein zunehmend alternierender Weg in G ausgehend von u, der eine Blüte B passiert. Es gibt zwei Fälle:

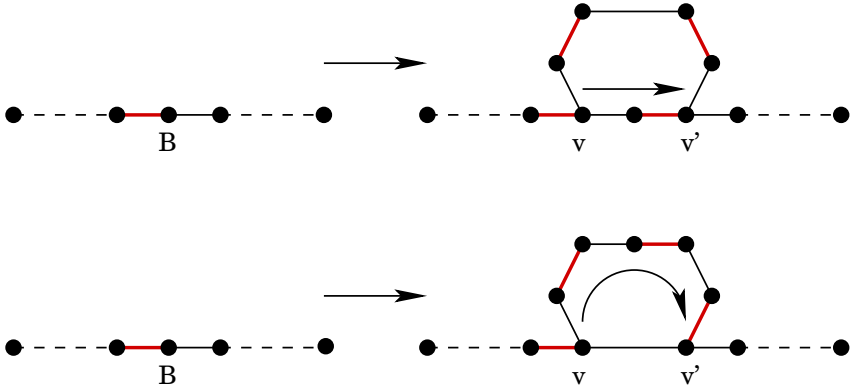
- P berührt die Blüte nur in einem Knoten. Dann bleibt der Pfad unverändert.



- P betritt die Blüte in einem Knoten v und verlässt sie wieder in einem anderen Knoten v'. In der Blüte gibt es zwei Pfade von v nach v': einen mit gerader Länge und einen mit ungerader Länge. Einer der beiden Pfade erhält die Parität des Abstands von u nach v'.



Expandieren von Blüten: Für den Weg durch eine Blüte gibt es zwei Möglichkeiten. Je nachdem, welche Parität notwendig ist, wird der korrekte Pfad gewählt.



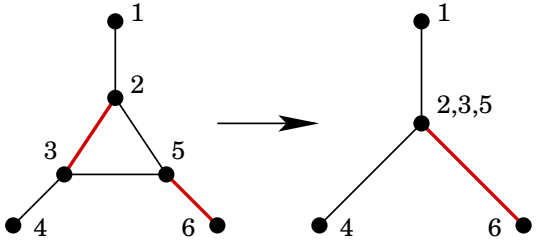
Laufzeit von Edmond's Algorithmus: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$

- Alle Operationen in einer Runde dauern Zeit $\mathcal{O}(\mathcal{E})$:
  - Alternierenden Weg suchen mittels Breitensuche von einem freien Knoten aus: $\mathcal{O}(\mathcal{V} + \mathcal{E})$
  - Auf einem Pfad können alle Blüten zusammen höchstens $\mathcal{E}$ Kanten besitzen. Das Schrumpfen einer Blüte ist abhängig von der Größe des Kreises. Demnach können alle Blüten in einem Durchlauf in $\mathcal{O}(\mathcal{E})$ Schritten geschrumpft werden.
- In jeder Runde wird das Matching größer, also gibt es höchstens $\mathcal{V}/2$ viele Runden.

Auch diese Laufzeit kann noch verbessert werden:

> S. Micali, V. Vazirani (1980): An $\mathcal{O}(\sqrt{\mathcal{V}} \cdot \mathcal{E})$ algorithm for finding maximum matching in general graphs. 21st Annual Symposium on Foundations of Computer Science: 17–27

Anmerkung: Es genügt nicht, zunächst alle Blüten zu suchen und zu schrumpfen, um anschließend eine Breitensuche durchzuführen:



Im obigen Beispiel enthält der geschrumpfte Graph keinen zunehmend alternierenden Pfad, wohl aber der Ausgangsgraph.

Übersicht

Sei $G = (V, E)$ ein ungerichteter Graph.

- Die Anzahl der Knoten in einer Maximum-Clique (also einer Clique mit maximaler Kardinalität) wird mit $\omega(G)$ bezeichnet und heißt *Cliquenzahl* (clique number).
- Seien $C_1, \ldots, C_k$ Cliquen in G, die G partitionieren, also $V = C_1 \cup \ldots \cup C_k$ und $C_i \cap C_j = \emptyset$ für alle $1 \leq i, j \leq k$ und $i \neq j$. Dann heißt $C_1, \ldots, C_k$ eine *Cliquen-Partitionierung* der Größe k.
- Die Größe der kleinstmöglichen Cliquen-Partitionierung wird mit $\theta(G)$ bezeichnet und heißt *Cliquen-Überdeckungszahl* (clique cover number).

Die rot gezeichneten Knoten stellen jeweils eine Maximum-Clique dar. Insgesamt stellen die farbigen Teilgraphen jeweils ein Minimum-Clique-Cover dar.

Sei $G = (V, E)$ ein ungerichteter Graph.

- Sei I eine Teilmenge der Knoten aus V, die paarweise nicht adjazent sind. Man nennt I eine *unabhängige Menge* (independent set) oder eine *stabile Menge* (stable set).
- Die Anzahl der Knoten einer unabhängigen Menge mit maximaler Kardinalität wird mit $\alpha(G)$ bezeichnet und heißt *Unabhängigkeitszahl* oder *Stabilitätszahl* (stability number).

Die blau gezeichneten Knoten stellen jeweils ein Maximum-Independent-Set dar.

Sei $G = (V, E)$ ein ungerichteter Graph.

- Seien $I_1, \ldots, I_k$ unabhängige Mengen von G. Stellen diese unabhängigen Mengen $I_1, \ldots, I_k$ eine Partitionierung des Graphen G dar, also $V = I_1 \cup \ldots \cup I_k$ und $I_i \cap I_j = \emptyset$ für alle $1 \leq i, j \leq k$ und $i \neq j$, so spricht man von einer *k-Färbung* (k-coloring) von G.
- Das minimale k, für welches eine k-Färbung von G existiert, wird mit $\chi(G)$ bezeichnet und heißt *Färbungszahl* oder *chromatische Zahl* (chromatic number).

Übung 35. *Begründen Sie, warum $\chi(G) \leq \Delta(G) + 1$ für jeden Graphen $G = (V, E)$ gilt, wobei $\Delta(G) := \max_{v \in V}\{deg_G(v)\}$ den maximalen Knotengrad von G bezeichnet.*

Allgemein gilt:

- $\omega(G) \leq \chi(G)$
  Alle Knoten einer Clique müssen mit verschiedenen Farben gefärbt sein.
- $\alpha(G) \leq \theta(G)$
  Alle Knoten einer unabhängigen Menge müssen in verschiedenen Cliquen einer Cliquen-Partitionierung liegen.
- $\omega(G) = \alpha(\overline{G})$
  Eine Clique maximaler Kardinalität in G entspricht einer unabhängigen Menge maximaler Kardinalität in $\overline{G}$.
- $\chi(G) = \theta(\overline{G})$
  Eine k-Färbung in G, also eine Partitionierung in k unabhängige Mengen, entspricht einer Cliquen-Partitionierung in $\overline{G}$.

Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißen *isomorph* zueinander, wenn eine bijektive Abbildung $f : V_1 \to V_2$ existiert, wobei $\{u, v\} \in E_1 \iff \{f(u), f(v)\} \in E_2$ gelten muss. Wir schreiben dann $G_1 \simeq G_2$.

Anmerkung[14]: The problem is not known to be solvable in polynomial time nor to be NP-complete.

Übung 36.

- *Geben Sie Graphen an, bei denen $\omega(G) < \chi(G)$ gilt.*
- *Geben Sie Graphen an, bei denen $\alpha(G) < \theta(G)$ gilt.*
- *Erstellen Sie den Komplementgraphen zu P_4, also einem Weg über vier Knoten.*
- *Erstellen Sie den Komplementgraphen zu C_5, also einem Kreis über fünf Knoten.*

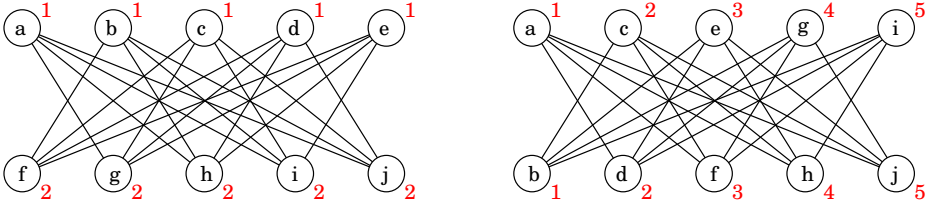
[14] `https://en.wikipedia.org/wiki/Graph_isomorphism_problem`

Greedy-Coloring: Sei $\sigma = (v_1, v_2, \ldots, v_n)$ eine Sortierung der Knoten.

for $i := 1$ **to** n **do**

color vertex v_i with smallest possible color

Eine solche Färbung kann deutlich mehr Farben benötigen als eine optimale Färbung, wie man am Beispiel des Crown-Graphen sieht:



Die Greedy-Färbung erfolgt in diesen Beispielen anhand der alphabetischen Reihenfolge der Knoten: Links reichen 2 Farben aus, rechts werden $n/2$ Farben benötigt.

Anmerkung: Mittels der Greedy-Färbung kann auch $\chi(G) \leq \Delta(G) + 1$ gezeigt werden.

Übung 37. *Welcher Zusammenhang besteht zwischen einem Minimum-Vertex-Cover und einem Maximum-Independent-Set?*

Übung 38. *Enthält ein Minimum-Clique-Cover immer eine maximal große Clique?*

Eine Grapheigenschaft A ist eine Untermenge der Menge aller Graphen, so dass für je zwei isomorphe Graphen G_1 und G_2 gilt: $G_1 \in A \iff G_2 \in A$.

Ein Graph G hat die Eigenschaft A, falls $G \in A$ ist.

- Eine Grapheigenschaft heißt *monoton*, falls für jeden Graphen mit dieser Eigenschaft gilt, dass auch jeder seiner Subgraphen diese Eigenschaft hat.
- Eine Grapheigenschaft heißt *hereditär*, falls für jeden Graphen mit dieser Eigenschaft gilt, dass auch jeder seiner induzierten Subgraphen diese Eigenschaft hat.

Jede monotone Eigenschaft ist auch hereditär. Aber nicht jede hereditäre Eigenschaft ist auch monoton: Die Eigenschaft eines Graphen, vollständig zu sein, ist hereditär, aber nicht monoton.

Übersicht

Ein zusammenhängender Graph $G = (V, E)$ mit $|E| = |V| - 1$ Kanten ist ein *Baum* und insbesondere kreisfrei. Ein kreisfreier Graph ist ein *Wald*.

Für Bäume und Wälder gilt: $\omega(G) = \chi(G) = 2$, falls $|E| > 0$

Übung 39. *Ist die Eigenschaft Baum monoton/hereditär?*

Übung 40. *Ist die Eigenschaft Wald monoton/hereditär?*

Ein Graph $G = (V, E)$ heißt *bipartit*, falls $V = A \cup B$ gilt mit $A \cap B = \emptyset$ und $\{u, v\} \in E \Rightarrow (u \in A, v \in B) \vee (v \in A, u \in B)$.

Ein Graph ist genau dann bipartit, wenn er keinen Kreis ungerader Länge enthält.

Für bipartite Graphen gilt: $\omega(G) = \chi(G) = 2$, falls $|E| > 0$

Übung 41. *Ist die Eigenschaft bipartit monoton/hereditär?*

Maximum-Independent-Set: Sei $G = (V, E)$ ein kreisfreier Graph.

Wähle iterativ einen Knoten v mit Grad 1, auch Blatt genannt.

- Sei $\{u, v\}$ die zu v inzidente Kante.
- Füge v zum Independent-Set hinzu.
- Entferne u und v und alle dazu inzidenten Kanten aus dem Graphen.

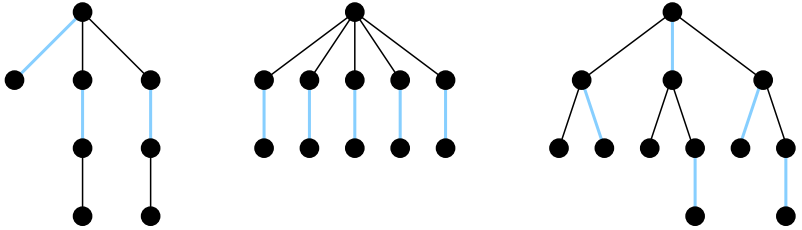
Füge schließlich alle verbleibenden isolierten Knoten dem Independent-Set hinzu.

Korrektheit: Wenn v ein Blatt ist, dann gibt es ein Max-Indep.-Set, das v enthält. Für das folgende *Austauschargument* sei S ein Max-Indep.-Set.

1. $v \in S \Rightarrow \checkmark$
2. $u \notin S, v \notin S \Rightarrow S \cup \{v\}$ ist ein Indep.-Set, aber S hat nicht maximale Größe $\lightning$
3. $u \in S, v \notin S \Rightarrow S \cup \{v\} - \{u\}$ ist ein Independent-Set gleicher Größe $\checkmark$

Minimum-Clique-Cover ist in polynomieller Zeit lösbar für C_3-freie Graphen, also insbesondere für Bäume und Wälder. Sei $G = (V, E)$ ein kreisfreier Graph.

- Bestimme ein Maximum-Matching $M = \{e_1, \ldots, e_k\} \subseteq E$. Da Bäume bzw. Wälder bipartite Graphen sind, geht das in Zeit $\mathcal{O}(\sqrt{\mathcal{V}} \cdot \mathcal{E})$.
- Es gilt $\omega(G) \leq 2$, denn jede Clique im kreisfreien Graphen G besteht aus maximal zwei Knoten. (Sonst wäre ein Kreis C_3 in G enthalten.)
- Das Minimum-Clique-Cover besteht also aus den Cliquen $e_i = \{u_i, v_i\}$ sowie aus den einzelnen Knoten, zu denen keine Matching-Kante inzident ist.



Es seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei disjunkte Graphen, also $V_1 \cap V_2 = \emptyset$.

- Die *disjunkte Vereinigung* von G_1 und G_2 ist definiert durch

$$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2).$$

- Die *disjunkte Summe* von G_1 und G_2 ist definiert durch

$$G_1 \times G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{\{v_1, v_2\} \mid v_1 \in V_1, v_2 \in V_2\}).$$

Ein Graph $G = (V, E)$ ist ein *Co-Graph*, falls er sich aus den folgenden drei Regeln aufbauen lässt:

- Der Graph mit genau einem Knoten (Bezeichnung $G = \bullet$) ist ein Co-Graph.
- Die disjunkte Vereinigung zweier Co-Graphen ist ein Co-Graph.
- Die disjunkte Summe zweier Co-Graphen ist ein Co-Graph.

Beobachtung: Ein Graph ist genau dann ein Co-Graph, wenn er keinen P_4 als induzierten Teilgraphen enthält.

Co-Graphen waren ursprünglich anders definiert:

- Der Graph mit genau einem Knoten (Bezeichnung $G = \bullet$) ist ein Co-Graph.
- Die disjunkte Vereinigung zweier Co-Graphen ist ein Co-Graph.
- Ist G ein Co-Graph, dann ist auch $\overline{G}$ ein Co-Graph.

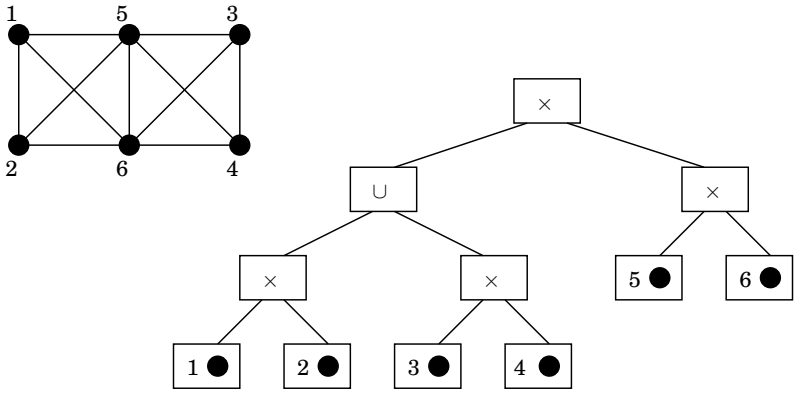
Beide Definitionen sind äquivalent. Die ursprüngliche Definition erklärt aber besser den Namen der Graphklasse: Complement-Reducible-Graph

Lemma: Für zwei Co-Graphen G_1 und G_2 gilt:

$$G_1 \times G_2 \;=\; \overline{\overline{G_1} \cup \overline{G_2}}$$

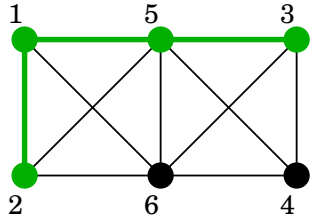
Übung 42. *Beweisen Sie das obige Lemma.*

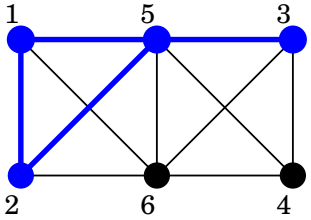
Der Aufbau eines Co-Graphen kann in einer Baumstruktur dargestellt werden: Links der Co-Graph, rechts der Co-Baum.



Übung 43. *Stellen Sie den Co-Graphen als Baum über die Operationen der alternativen Definition dar, also mittels disjunkter Vereinigung und Komplementbildung.*

Anmerkung: Der obige Co-Graph enthält den P_4 zwar als Teilgraphen, linkes Bild, aber nicht als induzierten Teilgraphen, rechtes Bild.





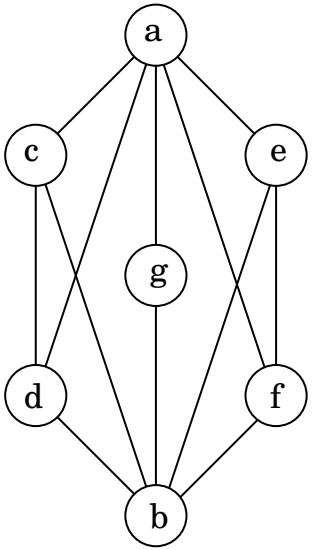
Satz: Zu einem gegebenen Graphen $G = (V, E)$ kann in Zeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$ entschieden werden, ob G ein Co-Graph ist und im positiven Fall auch ein Co-Baum für G bestimmt werden. (A linear recognition algorithm for cographs: Corneil, Perl, Stewart, 1985)

Erstellen eines Co-Baums: Sei G ein Co-Graph. Iteriere die folgenden Schritte, bis die Graphen nur noch aus einem Knoten bestehen.

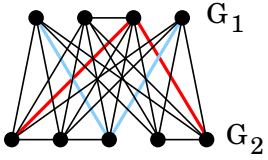
- Falls G nicht zusammenhängend ist, dann seien $G_1, \ldots, G_k$ seine Zusammenhangskomponenten. Erstelle rekursiv den Co-Baum für jede Komponente G_i und erstelle einen $\cup$-Knoten als Vorgängerknoten.
- Sonst betrachte den Komplementgraphen $\overline{G}$ und seine Zusammenhangskomponenten $\overline{G_1}, \ldots, \overline{G_k}$. Erstelle rekursiv den Co-Baum für jeden Komplementgraphen G_i und erstelle einen $\times$-Knoten als Vorgänger.

Dieser Algorithmus liefert z.B. für den P_4 kein Ergebnis, da der P_4 zusammenhängend ist, der Komplementgraph $\overline{P_4}$ aber auch, denn wir hatten bereits gesehen, dass der P_4 isomorph zu seinem Komplementgraphen ist: $P_4 \simeq \overline{P_4}$

Übung 44. *Erstellen Sie für den folgenden Co-Graphen den Co-Baum.*



Bemerkung: Bei einem zusammenhängenden Co-Graphen G ist die letzte Operation eine $\times$-Operation. Daher hat G einen Durchmesser von höchstens 2, der längste kürzeste Weg zwischen 2 Knoten hat also höchstens die Länge 2.



Anwendung von Co-Graphen:

- M. Hellmuth, M. Hernandez-Rosales, K.T. Huber, V. Moulton, P.F. Stadler, N. Wieseke: Orthology Relations, Symbolic Ultrametrics, and Cographs. Journal of Mathematical Biology 66 (2013)
- M. Hellmuth, P. Stadler, N. Wieseke: The mathematics of xenology: di-cographs, symbolic ultrametrics, 2-structures and tree-representable systems of binary relations. Journal of Mathematical Biology 75 (2017)
- N. Nojgaard, N. El-Mabrouk, D. Merkle, N. Wieseke, M. Hellmuth: Partial homology relations - satisfiability in terms of di-cographs. In: Proceedings of International Computing and Combinatorics Conference. LNCS 10976 (2018)

Im Folgenden sei G ein Co-Graph und $T = (V_T, E_T, w)$ ein Co-Baum zu G mit der Wurzel w. Für einen Knoten $v \in V_T$ sei T_v der Teilbaum von T mit Wurzel v und $G[v]$ sei der durch T_v definierte Co-Graph.

Maximum Independent Set:

- Für jedes Blatt v in T setze $\alpha(G[v]) = 1$.
- Für jeden mit $\cup$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\alpha(G[v]) = \alpha(G[v_1]) + \alpha(G[v_2])$
- Für jeden mit $\times$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\alpha(G[v]) = \max\{\alpha(G[v_1]), \alpha(G[v_2])\}$

Maximum Clique:

- Für jedes Blatt v in T setze $\omega(G[v]) = 1$.
- Für jeden mit $\cup$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\omega(G[v]) = \max\{\omega(G[v_1]), \omega(G[v_2])\}$
- Für jeden mit $\times$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\omega(G[v]) = \omega(G[v_1]) + \omega(G[v_2])$

minimale Färbung:

- Für jedes Blatt v in T setze $\chi(G[v]) = 1$.
- Für jeden mit $\cup$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\chi(G[v]) = \max\{\chi(G[v_1]), \chi(G[v_2])\}$
- Für jeden mit $\times$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\chi(G[v]) = \chi(G[v_1]) + \chi(G[v_2])$

Minimum Clique Cover:

- Für jedes Blatt v in T setze $\theta(G[v]) = 1$.
- Für jeden mit $\cup$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\theta(G[v]) = \theta(G[v_1]) + \theta(G[v_2])$
- Für jeden mit $\times$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\theta(G[v]) = \max\{\theta(G[v_1]), \theta(G[v_2])\}$

Die hier gezeigten Algorithmen haben Laufzeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$:

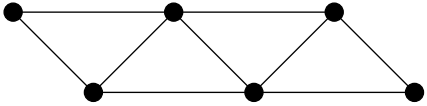
- Erstellung des Co-Baums in $\mathcal{O}(\mathcal{V} + \mathcal{E})$.
- Bottom-Up-Durchlauf der Knoten des Co-Baums in $\mathcal{O}(\mathcal{V} + \mathcal{E})$.

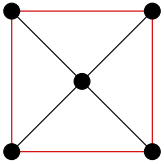
Übersicht

Ein Graph heißt *chordal*, falls er keinen induzierten Teilgraphen enthält, der isomorph zum Kreis C_k mit $k \geq 4$ ist. In anderen Worten: Jeder Kreis der Länge mindestens vier besitzt eine Sehne. Eine *Sehne* ist eine Kante zwischen zwei auf dem Kreis nicht benachbarten Knoten.

Beispiele:

- Der vollständige Graph K_n ist chordal.
- Ein Wald ist chordal.
- Der linke Graph ist chordal, der rechte nicht:





Übung 45. *Ist die Eigenschaft chordal monoton/hereditär?*

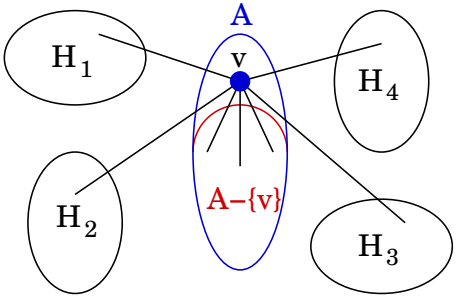
Sei $G = (V, E)$ ein Graph und $u, v \in V$ zwei Knoten.

- Eine Knotenmenge $A \subset V - \{u, v\}$ heißt *$u - v$-trennend*, falls u und v in G, aber nicht in $G - A$ durch einen Pfad verbunden sind.
- Eine Knotenmenge $A \subset V$ heißt *trennend*, falls es Knoten $u, v \in V$ gibt, so dass A $u - v$-trennend ist.

Lemma: Sei A eine inklusionsweise-minimal trennende Knotenmenge in einem zusammenhängenden Graphen $G = (V, E)$. Dann ist jeder Knoten in A zu jeder Komponente von $G - A$ verbunden.

Beweis: Da A minimal ist, gilt für jedes $v \in A$, dass der Graph $G - (A - \{v\})$ noch zusammenhängend ist. Da es zwischen den Komponenten $H_1, \ldots, H_k$ von $G - A$ keine Kanten gibt, sind die H_i's in $G - (A - \{v\})$ alle mit v verbunden, also insbesondere in G. (siehe auch nächste Folie)

noch zum Beweis:



- $G - (A - \{v\})$ ist noch zusammenhängend
- $G - A$ zerfällt in $H_1, \ldots, H_4$

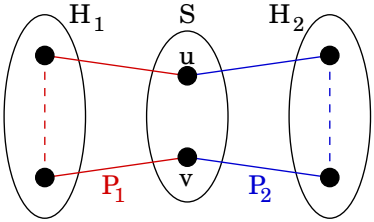
Definition: Wir bezeichnen die Menge der adjazenten Knoten eines Knotens u in einem Graphen $G = (V, E)$ als Nachbarschaft und schreiben $N_G(u) := \{v \mid \{u, v\} \in E\}$.

Falls aus dem Kontext heraus klar ist, welcher Graph G gemeint ist, so schreiben wir $N(u)$ anstelle von $N_G(u)$. Für $A \subseteq V$ definieren wir $N_G(A) := \bigcup_{u \in A} N_G(u)$.

Satz: Ein Graph ist chordal $\iff$ jede inklusionsweise-minimal trennende Knotenmenge induziert eine Clique. (Dirac, 1961)

$\Rightarrow$ Sei S eine solche trennende Menge in G.

- Es gibt mindestens zwei Komponenten H_1 und H_2 in $G - S$.
- Alle Knoten aus S haben Nachbarn in H_1 und H_2. (Lemma)
- Seien $u, v \in S$, und seien P_1, P_2 zwei $u - v$-Pfade **minimaler Länge**, die nur innere Knoten aus H_1 bzw. H_2 benutzen.



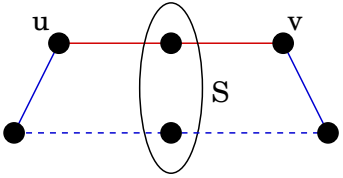
- Der Kreis $P_1 \cup P_2$ hat Länge $\geq$ vier und daher eine Sehne.
- Zwischen H_1 und H_2 gibt es keine Kanten, die Pfade P_1, P_2 haben **minimale Länge**, es gibt also keine Sehne innerhalb von H_1 oder H_2, also muss die Kante $\{u, v\} \in E$ existieren.

Fortsetzung

Satz: Ein Graph ist chordal $\iff$ jede inklusionsweise-minimal trennende Knotenmenge induziert eine Clique. (Dirac, 1961)

$\Leftarrow$ Sei C ein Kreis der Länge ≥ 4 in G und seien $u, v \in C$ zwei auf C nicht benachbarte Knoten.

- Ist $\{u, v\} \in E$, so gibt es eine Sehne. ✓
- Sonst sei S eine minimal $u - v$-trennende Knotenmenge.
- S enthält je einen Knoten auf beiden $u - v$-Pfaden in C.



- Da S eine Clique ist, sind diese durch eine Kante verbunden. ✓

Ein Knoten in einem Graphen heißt *Simplizialknoten*, falls seine Nachbarschaft vollständig verbunden ist, also eine Clique bildet.

Lemma: Jeder chordale Graph G besitzt einen Simplizialknoten, und falls der Graph nicht vollständig ist, besitzt er sogar **zwei nicht benachbarte** Simplizialknoten. (Dirac, 1961)

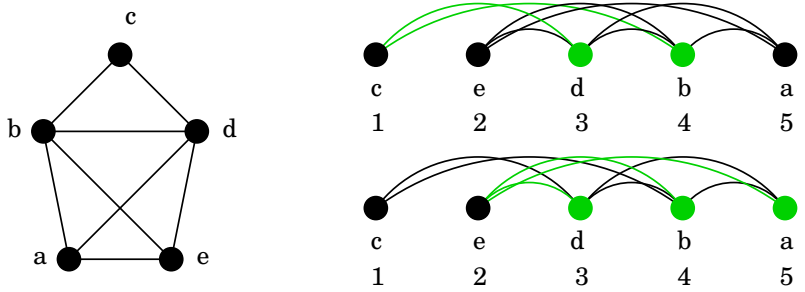
Beweis: Durch vollständige Induktion nach $n = |V|$.

- Aussage ist klar für $n = 1, 2, 3$ oder vollständige Graphen. ✓
- Sonst: G besitzt zwei nicht benachbarte Knoten u und v.
- Sei S eine inklusionsweise-minimal $u - v$-trennende Knotenmenge, $G - S$ zerfalle in Komponenten $H_1, \ldots, H_k$.
- S induziert Clique, also haben nach Induktionsannahme die chordalen Graphen $G[H_i \cup S]$ je einen Simplizialknoten in H_i:
  - Entweder gibt es zwei nicht benachbarte simpliziale Knoten, von denen einer in H_i liegt, da S eine Clique ist.
  - oder $G[H_i \cup S]$ ist Clique, also jeder Knoten aus H_i simplizial.
- Diese Simplizialknoten in H_i sind auch simplizial in G, weil für die Nachbarschaft $N(H_i)$ gilt: $N(H_i) \subseteq H_i \cup S$

Die Knoten eines chordalen Graphen $G = (V, E)$ mit n Knoten lassen sich wie folgt anordnen.

> Eine totale Ordnung $\sigma : V \to [n]$ heißt *perfektes Knoten-Eliminationsschema* PES, falls jeder Knoten $v \in V$ simplizial in $G[\{u \in V \mid \sigma(u) > \sigma(v)\}]$ ist.
>
> anders gesagt: Die Anordnung $(v_1, \ldots, v_n)$ der Knoten aus G heißt PES, wenn die Mengen $N_i := \{v_j \in N(v_i) \mid j > i\}$ Cliquen sind.

Beispiel:



Satz: Ein Graph ist genau dann chordal, wenn er ein perfektes Knoten-Eliminationsschema besitzt. (Fulkerson, Gross, 1965)

$\Rightarrow$ Sei G ein chordaler Graph und s ein Simplizialknoten.

- Das PES beginnt mit $(s, \ldots)$.
- Auch $G - \{s\}$ ist chordal. (weil hereditär)
- Wähle sukzessiv Simplizialknoten im restlichen Graphen aus und hänge den Knoten an das bisherige PES an.
- Falls keiner gefunden wird, ist der Graph nicht chordal.

$\Leftarrow$ Sei G ein Graph mit PES σ und sei C ein Kreis in G.

- Sei $u \in C$ der erste Knoten des Kreises bezüglich σ.
- Nach Definition des PES sind insbesondere seine beiden Nachbarn auf C verbunden.
- Damit hat der Kreis eine Sehne.

Die Existenz eines PES ist also hinreichend für die Chordalität eines Graphen.

Obige naive Implementierung zur Berechnung eines PES hat Laufzeit $\mathcal{O}(\mathcal{V}^2 \cdot \mathcal{E})$.
Mittels einer lexikographischen Breitensuche kann ebenfalls ein PES berechnet werden, aber effizienter.

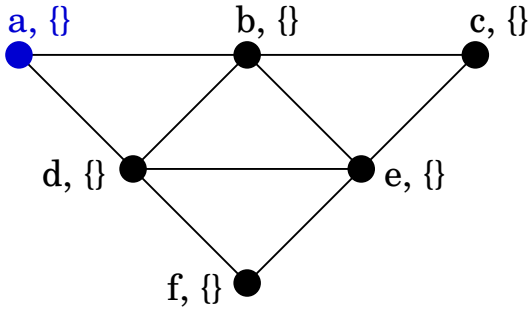
Lex-Bfs ist eine normale Breitensuche mit Kollisionsregel:

for all $v \in V$ **do**
    $label[v] := \emptyset$
for $i := n$ **downto** 1 **do**
    pick unnumbered vertex v with lexicographically largest label
    $\sigma(v) := i$
    for all unnumbered vertices w adjacent to v **do**
        append $label[w]$ by i

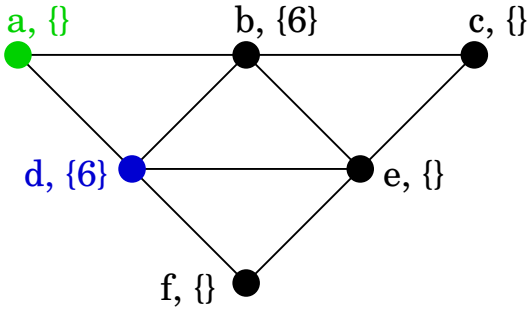
Es gilt $(a_1, \ldots, a_q) \leq_L (b_1, \ldots, b_p)$, falls es ein j gibt mit $a_i \leq b_i$ für $1 \leq i \leq j$ und ($q < p$ oder $a_{j+1} < b_{j+1}$) gilt.

So gilt bspw. $(6, 3, 2, 1) \leq_L (6, 4, 1)$ und $(6, 4, 2) \leq_L (6, 4, 2, 1)$.

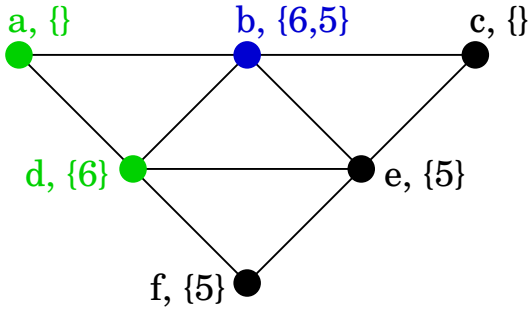
Beispiel: $i = 6$, wähle Knoten $a \to \sigma = (a)$



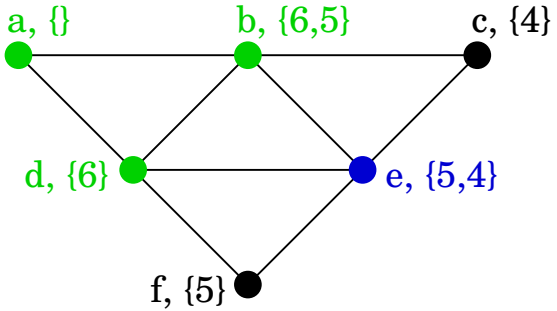
Beispiel: $i = 5$, wähle Knoten $d \to \sigma = (d, a)$



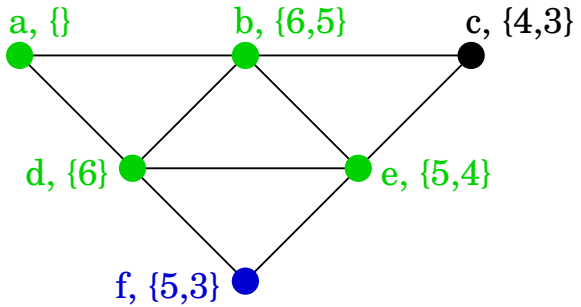
Beispiel: $i = 4$, wähle Knoten $b \to \sigma = (b, d, a)$



Beispiel: $i = 3$, wähle Knoten $e \to \sigma = (e, b, d, a)$



Beispiel: $i = 2$, wähle Knoten $f \rightsquigarrow \sigma = (f, e, b, d, a)$



Nach einem weiteren Schritt erhalten wir (c, f, e, b, d, a) als Ordnung. Man prüft leicht nach, dass diese Ordnung ein PES ist.

Satz: Der Algorithmus LEX-BFS liefert bei Anwendung auf einen Graphen G genau dann ein PES, wenn G chordal ist. (ohne Beweis)

Beobachtung: Jeder chordale Graph G ist *perfekt*, d.h. für alle induzierten Teilgraphen H von G gilt: $\omega(H) = \chi(H)$ oder äquivalent $\omega(G[V']) = \chi(G[V'])$ für alle $V' \subseteq V$

Beweis: Wir zeigen $\omega(G) = \chi(G)$ für alle chordalen Graphen G. Da jeder induzierte Teilgraph eines chordalen Graphen auch chordal ist, ist damit die Aussage gezeigt.

Vollständige Induktion über die Anzahl der Knoten:

- I.A.: Aussage ist klar für $\mathcal{V} = 1$. ✓
- I.S. für $\mathcal{V} > 1$: Sei $v \in V$ ein simplizialer Knoten.
  - Laut I.V. gilt $\omega(G - \{v\}) = \chi(G - \{v\})$, also reichen $\omega(G) \geq \omega(G - \{v\})$ viele Farben, um $G - \{v\}$ zu färben.
  - Da $N_G(v) \cup \{v\}$ eine Clique ist, also insbesondere $|N_G(v)| \leq \omega(G) - 1$ ist, kann $G - \{v\}$ rekursiv mit $\omega(G)$ gefärbt werden, aber es bleibt eine Farbe für v frei.
  - Also existiert eine Färbung von G mit $\omega(G)$ Farben, also gilt $\chi(G) \leq \omega(G)$.
  - Da für alle Graphen G stets $\omega(G) \leq \chi(G)$ gilt, ist damit $\omega(G) = \chi(G)$ gezeigt.

Übung 46. *Warum wird für perfekte Graphen G als Eigenschaft $\omega(H) = \chi(H)$ für alle induzierten Teilgraphen H von G gefordert? Würde $\omega(G) = \chi(G)$ nicht ausreichen?*

Maximum Clique:

$K := \emptyset$
for all node u in order of σ **do**
  compute $X_u := \{v \in V \mid \{u, v\} \in E, \sigma(v) > \sigma(u)\}$
  if $|K| < |\{u\} \cup X_u|$ **then**
    $K := \{u\} \cup X_u$
output K

Optimalität? Irgendein Knoten u einer Max-Clique hat kleinsten Wert $\sigma(u)$ im PES σ.

Maximum Independent Set:

- sei y_1 der erste Knoten bezüglich σ
- induktiv: sei y_i der erste auf y_{i-1} folgende Knoten in σ, der nicht in der Menge $X_{y_1} \cup \ldots \cup X_{y_{i-1}}$ ist
- die Knoten $y_1, \ldots, y_t$ bilden eine unabhängige Menge

Optimalität $\rightarrow$ siehe nächste Folie

Minimum Clique Cover:

Die Mengen $\{y_i\} \cup X_{y_i} - X_{y_1} - \ldots - X_{y_{i-1}}$ bilden eine minimale Cliquen-Überdeckung.

Optimalität gilt wegen $t \leq \alpha(G) \leq \theta(G) \leq t$.

Minimum Coloring:

Betrachte die Knoten in der Reihenfolge von $\sigma = (v_1, \ldots, v_n)$.

- Färbe v_n mit 1.
- Für $i = n-1, \ldots, 1$ färbe v_i mit der kleinstmöglichen Zahl.

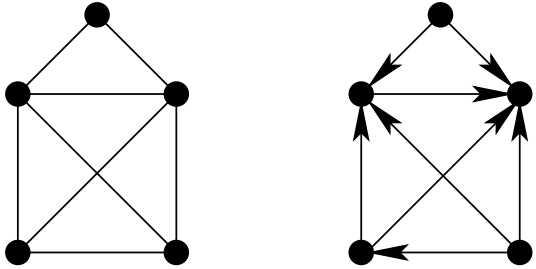
Ein Greedy-Coloring liefert eine optimale Färbung:

- Da $v_i \cup (N_G(v_i) \cap \{v_{i+1}, \ldots, v_n\}) =: C_i$ eine Clique bilden, gilt $|C_i| \leq \omega(G)$.
- Also gilt $|N_G(v_i) \cap \{v_{i+1}, \ldots, v_n\}| \leq \omega(G) - 1$ und v_i kann mit einer der Farben aus der Menge $\{1, \ldots, \omega(G)\}$ gefärbt werden.

Übersicht

Ein gerichteter Graph $G = (V, E)$ heißt *transitiv*, wenn für je drei Knoten $u, v, w \in V$ gilt: Falls $(u, v) \in E$ ist und $(v, w) \in E$ ist, dann ist auch $(u, w) \in E$.

Ein ungerichteter Graph heißt *Vergleichbarkeitsgraph* (comparability graph), wenn eine Orientierung der Kanten existiert, so dass der orientierte Graph transitiv ist.

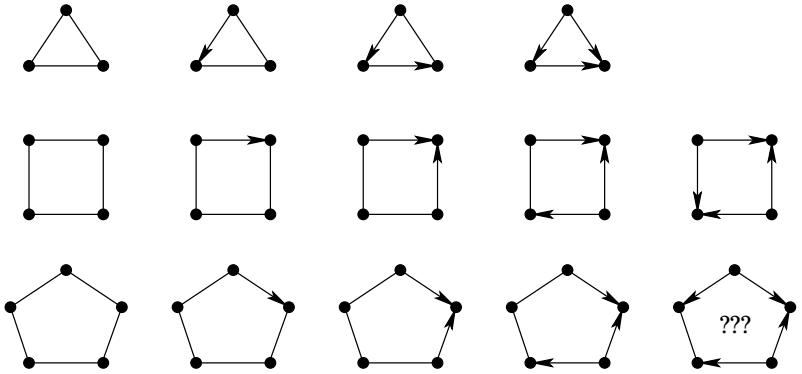


Links Vergleichbarkeitsgraph, rechts eine mögliche transitive Orientierung dazu.

Frage: Sind die Graphen $K_3 = C_3$, C_4 oder C_5 Vergleichbarkeitsgraphen?

Um eine Antwort zu finden, orientieren wir zunächst eine beliebige Kante. Da diese Graphen alle „symmetrisch“ sind, spielt die gewählte Richtung der Kante keine Rolle.

Dann ergibt sich nach und nach für einige der anderen Kanten ganz automatisch eine „notwendige“ Orientierung.



$\Rightarrow$ Der C_5 ist kein Vergleichbarkeitsgraph.

Frage: Ist die Eigenschaft *transitiv orientierbar* monoton/hereditär?

Antwort: Betrachte eine transitive Orientierung $G' = (V, A)$ des Graphen $G = (V, E)$.

- Die Eigenschaft *transitiv orientierbar* ist nicht monoton.

  Für $(u, v) \in A$ und $(v, w) \in A$ muss auch $(u, w) \in A$ gelten. Wird aber gerade die Kante (u, w) aus G' entfernt, dann ist die Orientierung nicht mehr transitiv. Die Eigenschaft gilt also nicht für beliebige Teilgraphen.
- Die Eigenschaft *transitiv orientierbar* ist hereditär.

  Betrachte einen beliebigen Knoten $u \in V$. Wäre $G' - \{u\}$ nicht transitiv, dann würden Kanten (a, b) und (b, c) existieren, für die (a, c) nicht vorhanden ist. Allerdings sind die einzigen Kanten, die in $G - \{u\}$ nicht vorhanden sind diejenigen, deren Start- oder Endknoten u ist. Also war schon G nicht transitiv. $\lightning$

Übung 47. *Wie beurteilen Sie die obigen Beweise?*

Wir führen für diesen Abschnitt eine andere Definition von Graphen ein, die auf M.C. Golumbic zurück geht:

- Ein Graph $G = (V, E)$ besteht aus einer anti-reflexiven binären Relation E über einer endlichen Knotenmenge V. Die Elemente von E bestehen aus geordneten Paaren von Knoten.

  Alle Graphen sind also Schleifen-frei und haben keine mehrfachen Kanten.
- Wir definieren $ab \in E^{-1} \iff ba \in E$.
- Ein Graph ist ungerichtet, wenn $E = E^{-1}$ gilt, d.h. ein ungerichteter Graph besteht aus gerichteten Kanten, aber zu jeder Kante gibt es eine entgegengesetzt gerichtete Kante.

Ein ungerichteter Graph $G = (V, E)$ heißt Vergleichbarkeitsgraph, wenn es einen Graphen (V, F) gibt, so dass gilt:

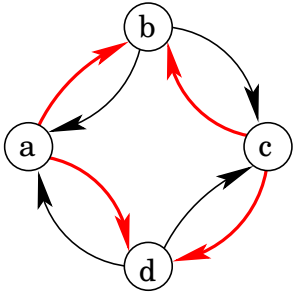
$$F \cap F^{-1} = \emptyset \qquad F \cup F^{-1} = E \qquad F^2 \subseteq F$$

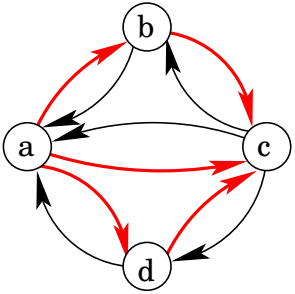
wobei $F^2 = \{ac \mid ab, bc \in F\}$ definiert sei.

Beispiel: ungerichteter Graph $G = (V, E)$

schwarze Kanten: F
rote Kanten: F^{-1}

$E = F \cup F^{-1}$
$F \cap F^{-1} = \emptyset$





$F^2 = \emptyset$

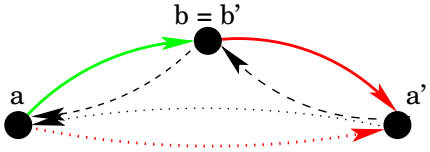
$F^2 = \{ca\}$

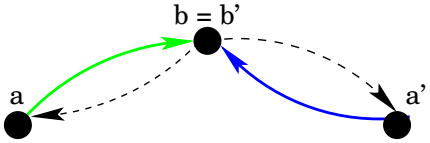
Definiere Relation Γ auf Knotenmenge E eines ungerichteten Graphen $G = (V, E)$ wie folgt. Sei $ab \in E$ und $a'b' \in E$.

$$ab \; \Gamma \; a'b' \qquad \Longleftrightarrow \qquad (a = a' \wedge bb' \notin E) \text{ oder } (b = b' \wedge aa' \notin E).$$

Wir sagen: ab fordert $a'b'$.

zweiter Fall: Würde ab und $b'a'$ mit $b = b'$ als Richtung der Kanten gewählt, müsste auch aa' gerichtet werden, was aber nicht möglich ist, da $aa' \notin E$ gilt.

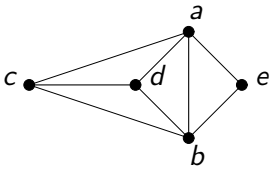


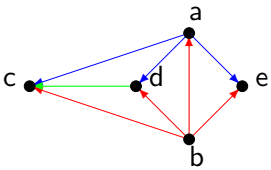


Die reflexive, transitive Hülle Γ^* von Γ ist eine Äquivalenzrelation auf E und partitioniert E in *Implikationsklassen* von G. Zwei Kanten ab und cd liegen genau dann in einer Implikationsklasse, wenn gilt: $ab = a_0b_0 \; \Gamma \; a_1b_1 \; \Gamma \ldots \Gamma \; a_kb_k = cd$ mit $k \geq 0$.

Bezeichne $\Phi(G)$ die Menge der Implikationsklassen von G, und bezeichne $\hat{A} = A \cup A^{-1}$ den symmetrischen Abschluss von A.

Beispiel:





$$A_1 = \{ab\} \qquad A_1^{-1} = \{ba\} \qquad \hat{A}_1 = A_1 \cup A_1^{-1}$$
$$A_2 = \{cd\} \qquad A_2^{-1} = \{dc\} \qquad \hat{A}_2 = A_2 \cup A_2^{-1}$$
$$A_3 = \{ac, ad, ae\} \qquad A_3^{-1} = \{ca, da, ea\} \qquad \hat{A}_3 = A_3 \cup A_3^{-1}$$
$$A_4 = \{bc, bd, be\} \qquad A_4^{-1} = \{cb, db, eb\} \qquad \hat{A}_4 = A_4 \cup A_4^{-1}$$

$\Phi(G) = \{\hat{A}_1, \hat{A}_2, \hat{A}_3, \hat{A}_4\}$ ist Menge der Implikationsklassen von G

Sei $G = (V, E)$ ein ungerichteter Graph.

- Eine Partition der Kantenmenge $E = \hat{B}_1 \uplus \ldots \uplus \hat{B}_k$ heißt *Decomposition* (Zerlegung) von E, wenn B_i eine Implikationsklasse von $\hat{B}_i \uplus \ldots \uplus \hat{B}_k$ darstellt.
- Eine Menge von Kanten $\{(x_1, y_1), \ldots, (x_k, y_k)\}$ heißt *Dekompositionsschema* für G, wenn eine Dekomposition $E = \hat{B}_1 \uplus \ldots \uplus \hat{B}_k$ existiert mit $(x_i, y_i) \in B_i$.

Berechnung eines Dekompositionsschemas für $G = (V, E)$:

S0 Initial sei $i := 1$ und $E_i := E$

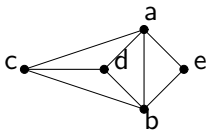
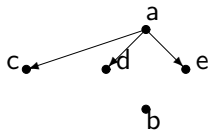
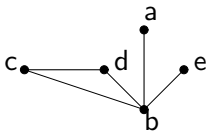
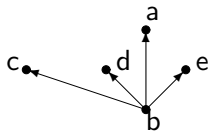
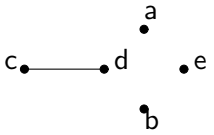
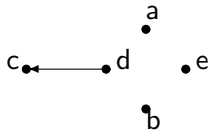
S1 Wähle eine Kante $e_i = (x_i, y_i) \in E_i$

S2 Zähle die Implikationsklasse B_i von E_i auf, die (x_i, y_i) enthält

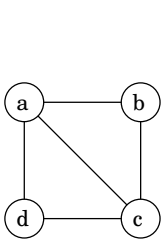
S3 Definiere $E_{i+1} := E_i \backslash \hat{B}_i$

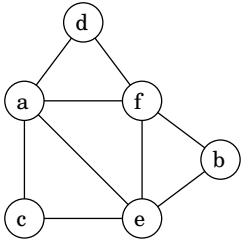
S4 Falls $E_{i+1} \neq \emptyset$, dann $i := i + 1$ und Sprung nach S1

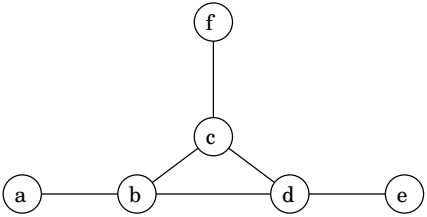
Laufzeit: $\mathcal{O}(\delta \cdot \mathcal{E})$, wobei δ den maximalen Knotengrad bezeichnet.

i	$G_i = (V, E_i)$	(x_i, y_i)	(V, B_i)
1		ac	
2		bc	
3		dc	

Übung 48. *Bestimmen Sie mit dem obigen Algorithmus eine Dekomposition, also eine transitive Orientierung der Kanten für den Graphen links.*







Übung 49. *Wenden Sie den obigen Algorithmus zur Berechnung einer Dekomposition auf den mittleren und rechten Graphen an. Gehören die dargestellten Graphen zur Klasse der Vergleichbarkeitsgraphen?*

Der folgende Satz zeigt, dass der Algorithmus zur Berechnung einer Dekomposition korrekt ist.

Satz: Sei $G = (V, E)$ ein ungerichteter Graph mit Dekomposition $E = \hat{B}_1 \uplus \ldots \uplus \hat{B}_k$, dann sind folgende Aussagen äquivalent:

- G ist ein Vergleichbarkeitsgraph
- $A \cap A^{-1} = \emptyset$ für alle Implikationsklassen A von E
- $B_i \cap B_i^{-1} = \emptyset$ für $i = 1, 2, \ldots, k$ (ohne Beweis)

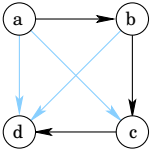
Anmerkung: Eine transitive Orientierung ist kreisfrei! Angenommen, es gäbe einen Kreis $C = (v_1, \ldots, v_n)$ mit den Kanten $v_1v_2, v_2v_3, \ldots, v_nv_1$. Dann gilt:

- $v_1v_3 \in F$, weil $v_1v_2 \in F$ und $v_2v_3 \in F$
- $v_1v_4 \in F$, weil $v_1v_3 \in F$ und $v_3v_4 \in F$
- $v_1v_5 \in F$, weil $v_1v_4 \in F$ und $v_4v_5 \in F$ usw.
- bis schließlich gilt: $v_1v_n \in F$, weil $v_1v_{n-1} \in F$ und $v_{n-1}v_n \in F$

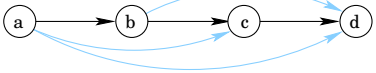
↯ $v_1v_n \in F$ und $v_nv_1 \in F$

Sei $G = (V, E)$ ein Vergleichbarkeitsgraph. (ab hier alte Definition von Graphen)

Betrachte eine transitive Orientierung $G' = (V, E')$ des Graphen:



- Rechts: Alle Knoten auf einem Weg stellen eine Clique dar.
- Unten: G' ist kreisfrei, daher kann eine topologische Sortierung berechnet werden. Werden die Knoten so angeordnet, dass alle Kanten nach rechts laufen, dann erkennt man den Weg.



Maximum Clique:

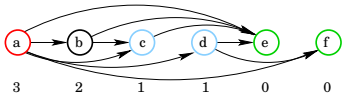
- berechne eine transitive Orientierung $G' = (V, E')$
- berechne eine topologische Sortierung $(v_n, \ldots, v_1)$ zu G'
- bestimme induktiv einen längsten Weg: $dist(v_1) := 0$, dann

$$dist(v_i) := 1 + \max_{j=1,\ldots,i-1} \{dist(v_j) \mid (v_i, v_j) \in E'\}$$

Minimum Coloring: Seien $dist(v_i)$ die zuvor berechneten Weglängen in G'.

- Wenn $dist(v_i) \leq dist(v_j)$ gilt, dann gibt es keine Kante (v_i, v_j), da sonst $dist(v_i) > dist(v_j)$ gelten müsste.

$\rightarrow$ Knoten mit gleicher Distanz bilden also ein Independent-Set und können gleich gefärbt werden.



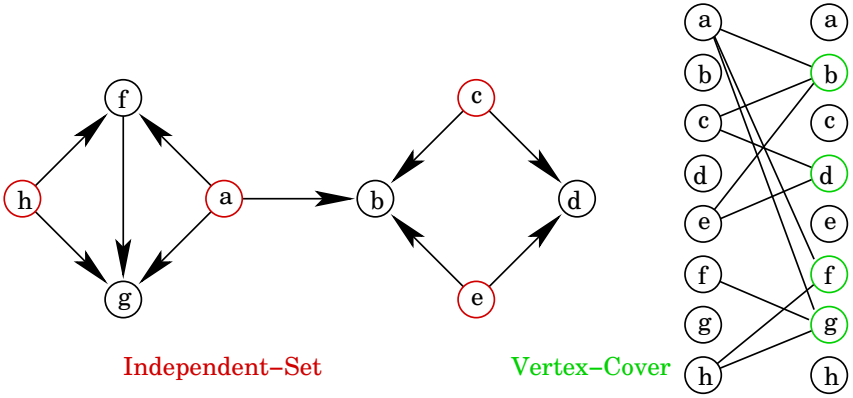
Mit den letzten beiden Aussagen haben wir gezeigt, dass $\omega(G) = \chi(G)$ für alle Vergleichbarkeitsgraphen G gilt. Da jeder induzierte Teilgraph eines Vergleichbarkeitsgraphen auch ein Vergleichbarkeitsgraph ist (hereditäre Grapheigenschaft), ist damit die folgende Aussage gezeigt.

Satz: Jeder Vergleichbarkeitsgraph G ist *perfekt*, d.h. für alle induzierten Teilgraphen H von G gilt: $\omega(H) = \chi(H)$ oder äquivalent $\omega(G[V']) = \chi(G[V'])$ für alle $V' \subseteq V$

Maximum Independent Set: Sei $V = \{v_1, \ldots, v_n\}$.

- berechne eine transitive Orientierung $G' = (V, E')$
- berechne einen bipartiten Graphen $G'' = (A \cup B, E'')$ mit
  - $A = \{x_i \mid v_i \in V\}$
  - $B = \{y_i \mid v_i \in V\}$
  - $E'' = \{\{x_i, y_j\} \mid (v_i, v_j) \in E'\}$ (siehe auch nächste Folie)
- berechne ein Minimum-Vertex-Cover T in dem bipartiten Graphen G''
- dann ist $S := V - T$ ein Maximum-Independent-Set

Beispiel:

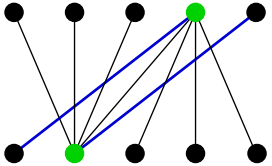


Frage: Wie berechnet man ein Minimum Vertex Cover auf bipartiten Graphen? Allgemein ist das Problem NP-vollständig.

Ein Minimum-Vertex-Cover auf bipartiten Graphen kann mittels eines Maximum-Matchings berechnet werden. (König, 1931)

Sei M ein Maximum-Matching auf dem bipartiten Graphen G.

Klar: Ein Vertex-Cover für G hat mindestens $|M|$ viele Knoten, da jede Matching-Kante durch das Vertex-Cover abgedeckt sein muss.

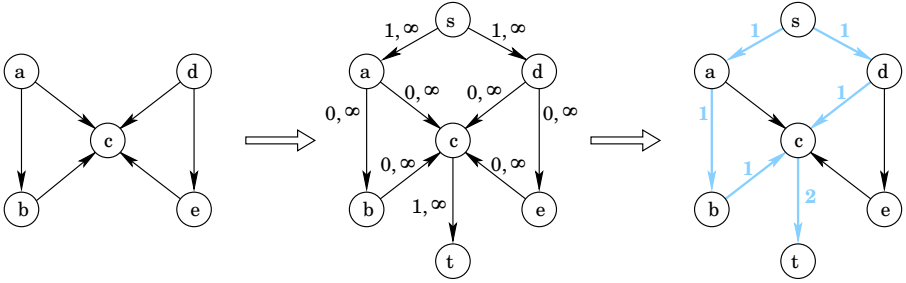


Die blauen Kanten stellen ein Maximum-Matching dar, die grünen Knoten ein Minimum-Vertex-Cover.

König 1931: In einem bipartiten Graphen ist die Größe eines größten Matchings gleich der Größe einer minimalen Knotenüberdeckung. (Beweis über alternierende Pfade)

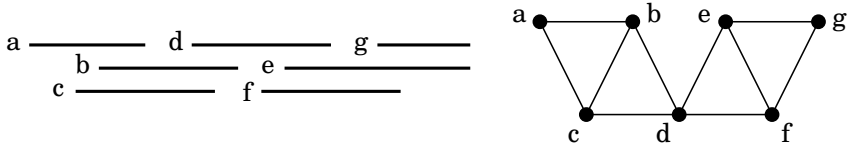
Minimum Clique Cover:

- Betrachte eine transitive Orientierung des Graphen.
- Füge eine Quelle s und eine Senke t hinzu; füge Kanten (s, x) und (y, t) hinzu, wenn x keine einlaufende und y keine auslaufende Kante besitzt.
- Weise jeder neuen Kante die minimale Kapazität 1 zu.
- Berechne einen Minimum-Fluss für das so erzeugte Netzwerk.
- Die Größe eines Minimum-Clique-Covers entspricht dem minimalen Fluss.



Intervallgraphen:

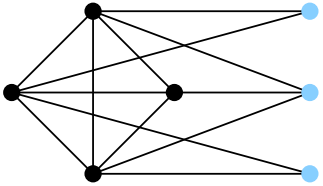
Sei $\mathcal{I} = \{I_1, \ldots, I_n\}$ eine Menge von Intervallen und sei $G = (V, E)$ ein Graph. Der Graph G heißt *Intervallgraph*, wenn $I_u \cap I_v \neq \emptyset \iff \{u, v\} \in E$ gilt. In diesem Fall nennt man $\mathcal{I}$ ein *Intervallmodell* für G.



Satz: Ein Graph G ist genau dann ein Intervallgraph, wenn G ein chordaler Graph und $\overline{G}$ ein Vergleichbarkeitsgraph ist. (ohne Beweis)

Split-Graphen:

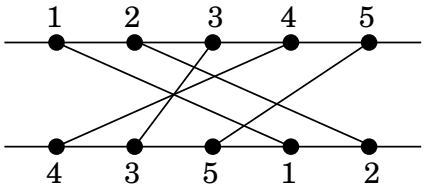
Ein Graph $G = (V, E)$ ist ein *Split-Graph*, wenn $V = A \cup B$ in eine Clique A und ein Independent-Set B mit $A \cap B = \emptyset$ partitioniert werden kann.

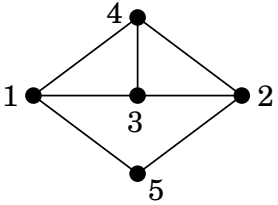


Satz: Ein Graph G ist genau dann ein Split-graph, wenn G und $\overline{G}$ beides chordale Graphen sind. (ohne Beweis)

Permutationsgraphen:

Ein Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ ist ein Permutationsgraph, wenn eine Permutation $\pi = (\sigma_1, \ldots, \sigma_n)$ existiert, sodass die Kante $\{v_i, v_j\} \in E$ existiert, genau dann wenn $i < j$ und $\sigma_i > \sigma_j$ gilt.





Satz: Ein Graph G ist genau dann ein Permutationsgraph, wenn G und $\overline{G}$ beides Vergleichbarkeitsgraphen sind. (ohne Beweis)

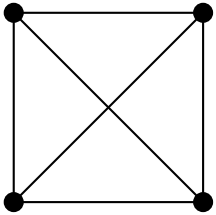
Übung 50.

1. *Geben Sie einen chordalen Graphen an, der kein Vergleichbarkeitsgraph ist.*
2. *Geben Sie einen chordalen Graphen an, der kein Intervallgraph ist.*
3. *Geben Sie einen chordalen Graphen an, der auch ein Intervallgraph ist.*
4. *Geben Sie einen Vergleichbarkeitsgraphen an, der nicht chordal ist.*
5. *Geben Sie einen Vergleichbarkeitsgraphen an, der kein Intervallgraph ist.*
6. *Geben Sie einen Vergleichbarkeitsgraphen an, der auch chordal ist.*
7. *Geben Sie einen Vergleichbarkeitsgraphen an, der auch ein Intervallgraph ist.*
8. *Geben Sie einen Vergleichbarkeitsgraphen an, der auch chordal ist, aber kein Intervallgraph ist.*
9. *Zeigen Sie, dass jeder Co-Graph auch ein Vergleichbarkeitsgraph ist.*

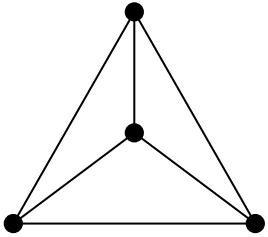
Übersicht

Definition: Ein Graph, der kreuzungsfrei in der Ebene gezeichnet werden kann, heißt planar. $\rightarrow$ planare Einbettung des Graphen

planarer Graph K_4:

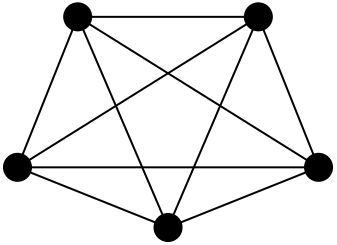


planare Einbettung des K_4:

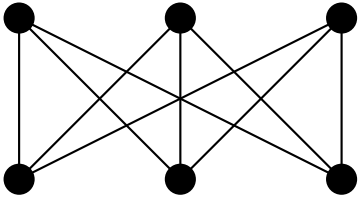


Alle einfachen Graphen mit höchstens 4 Knoten sind planar!

K_5:



$K_{3,3}$:

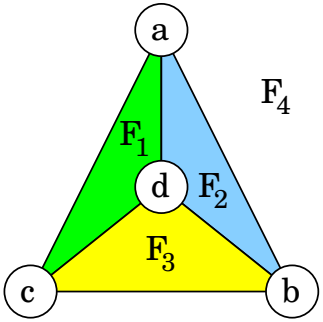


allgemein:

- K_n bezeichnet den vollständigen Graphen mit n Knoten
- $K_{n,m}$ bezeichnet den vollständig bipartiten Graphen $G = (V, E)$ mit $V = A \cup B$, $E = \{\{u, v\} \mid u \in A, v \in B\}$, $A \cap B = \emptyset$, $|A| = n$ und $|B| = m$.

Eulersche Polyeder-Formel: Für einen einfachen, zusammenhängenden, planaren Graphen $G = (V, E)$, der kreuzungsfrei in der Ebene gezeichnet ist, gilt $\mathcal{V} - \mathcal{E} + f = 2$, wobei f die Anzahl Flächen (faces) bezeichnet.

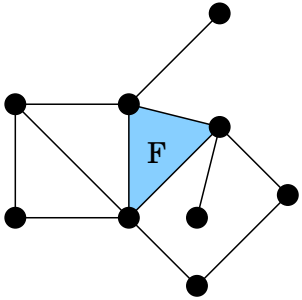
Beweis:



- Für einen Baum gilt: $\mathcal{E} = \mathcal{V} - 1$ und $f = 1$. Also gilt $\mathcal{V} - \mathcal{E} + f = 2$.
  An dieser Stelle im Beweis geht die Voraussetzung *zusammenhängend* ein.
- Entferne nacheinander aus jedem Kreis in G je eine Kante: Jedesmal wird $\mathcal{E}$ und f um 1 kleiner, die Differenz $\mathcal{V} - \mathcal{E} + f$ bleibt konstant. □

Korollar 1: Für einen planaren Graphen $G = (V, E)$ gilt: $\mathcal{E} \leq 3 \cdot \mathcal{V} - 6$

Beweis: Betrachte eine planare Einbettung von G mit f Flächen.



- Jede Kante e ist die Grenze von höchstens 2 Flächen.
  $\rightarrow$ p Kanten begrenzen maximal $2p$ Flächen.
- Dabei wird jede Fläche F mindestens dreimal gezählt, da F durch mindestens drei Kanten begrenzt ist. Beachte die Voraussetzung *einfacher Graph*.

$\rightarrow f \leq \frac{2 \cdot \mathcal{E}}{3} \iff 3 \cdot f \leq 2 \cdot \mathcal{E}$

Nach Eulers Polyeder-Formel gilt $\mathcal{V} - \mathcal{E} + f = 2$. Multiplizieren wir beide Seiten der Polyeder-Formel mit 3, dann erhalten wir:

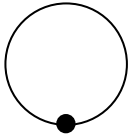
$$3 \cdot \mathcal{V} - 3 \cdot \mathcal{E} + 3 \cdot f = 6 \iff 3 \cdot f = 6 - 3 \cdot \mathcal{V} + 3 \cdot \mathcal{E}$$

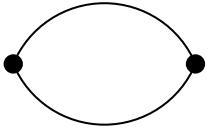
Jetzt nutzen wir die Ungleichung $3 \cdot f \leq 2 \cdot \mathcal{E}$ als Abschätzung in der modifizierten Polyeder-Formel $3 \cdot f = 6 - 3 \cdot \mathcal{V} + 3 \cdot \mathcal{E}$ und erhalten:

$$3 \cdot f \;=\; 6 - 3 \cdot \mathcal{V} + 3 \cdot \mathcal{E} \;\leq\; 2 \cdot \mathcal{E} \;\Longrightarrow\; \mathcal{E} \;\leq\; 3 \cdot \mathcal{V} - 6 \quad \square$$

Anmerkung: Planare Graphen $G = (V, E)$ sind dünne Graphen, es gilt also $\mathcal{E} \in \mathcal{O}(\mathcal{V})$.

Obiges Korollar gilt nur für einfache Graphen. Würden wir auch Schlingen oder Mehrfachkanten zulassen, dann könnte eine Fläche von weniger als drei Kanten begrenzt werden.





Korollar 2: Für einen bipartiten, einfachen, planaren Graphen $G = (V, E)$ gilt: $\mathcal{E} \leq 2 \cdot \mathcal{V} - 4$

Beweis: In bipartiten Graphen hat jeder Kreis eine gerade Länge. Also muss bei bipartiten Graphen sogar $4 \cdot f \leq 2 \cdot \mathcal{E}$ gelten, denn jede Fläche F wird dort mindestens viermal gezählt.

Nach der Eulerschen Polyeder-Formel gilt $\mathcal{V} - \mathcal{E} + f = 2$. Multiplizieren wir beide Seiten der Gleichung mit 4 erhalten wir:

$$4 \cdot \mathcal{V} - 4 \cdot \mathcal{E} + 4 \cdot f \;=\; 8 \iff 4 \cdot f \;=\; 8 - 4 \cdot \mathcal{V} + 4 \cdot \mathcal{E}$$

Nutzen wir $4 \cdot f \leq 2 \cdot \mathcal{E}$, dann erhalten wir:

$$4 \cdot f \;=\; 8 - 4 \cdot \mathcal{V} + 4 \cdot \mathcal{E} \;\leq\; 2 \cdot \mathcal{E} \implies 2 \cdot \mathcal{E} \;\leq\; 4 \cdot \mathcal{V} - 8$$

Dividieren wir beide Seiten durch 2, dann erhalten wir die zu beweisende Aussage. □

Satz: Der K_5 ist nicht planar.

Beweis: Im K_5 ist jeder der fünf Knoten mit jedem der vier anderen Knoten durch eine Kante verbunden. Da wir dabei jede Kante doppelt zählen, muss es also $\frac{4 \cdot 5}{2} = 10$ Kanten geben.

Nach Korollar 1 darf es aber höchstens $3 \cdot \mathcal{V} - 6 = 9$ Kanten geben. □

Satz: Jeder planare Graph besitzt einen Knoten v, der höchstens den Grad 5 hat.

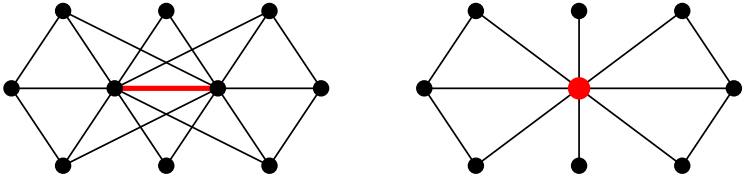
Beweis: Hätte jeder Knoten einen Knotengrad von mindestens 6, dann gäbe es mindestens $\frac{6 \cdot \mathcal{V}}{2} = 3 \cdot \mathcal{V}$ viele Kanten in dem Graphen, was nach Korollar 1 nicht sein kann. □

Satz: Der $K_{3,3}$ ist nicht planar.

Beweis: Im $K_{3,3}$ ist jeder Knoten mit drei anderen Knoten über eine Kante verbunden, es gibt also $\frac{6 \cdot 3}{2} = 9$ Kanten, nach Korollar 2 dürfte es aber nur $2 \cdot \mathcal{V} - 4 = 8$ Kanten geben. □

Wir wollen im Folgenden zu einem Graphen G testen, ob G planar ist. Beim Test spielen die Graphen K_5 und $K_{3,3}$ eine wichtige Rolle.

Kantenkontraktion:

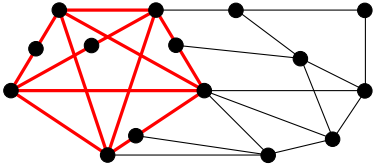


Ein Graph, der durch Löschen von Knoten oder Kanten oder durch Kantenkontraktionen aus G entsteht, heißt Graph-Minor von G.

Satz von Wagner: Jeder einfache Graph ist genau dann planar, wenn er weder den K_5 noch den $K_{3,3}$ als Minor enthält. (ohne Beweis)

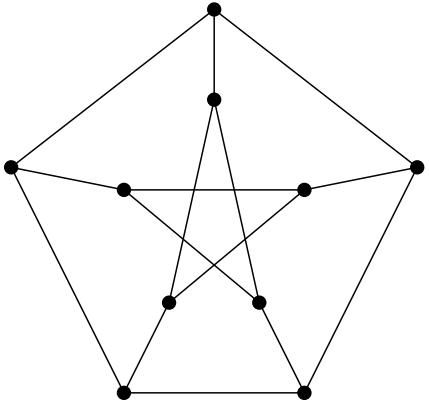
Ein Graph H ist eine *Unterteilung* von G, wenn H aus G entsteht, indem Kanten von G durch einfache Wege über neu eingefügte Knoten ersetzt werden:

Alle neu eingefügten Knoten haben also den Grad 2. Ein Teilgraph G' eines gegebenen Graphen G wird *Kuratowski-Teilgraph* genannt, wenn G' eine Unterteilung des K_5 oder des $K_{3,3}$ ist.



Satz von Kuratowski: Jeder einfache Graph ist genau dann planar, wenn er keinen Kuratowski-Teilgraphen enthält, wenn er also keine Unterteilung des K_5 oder des $K_{3,3}$ als Teilgraphen enthält. (ohne Beweis)

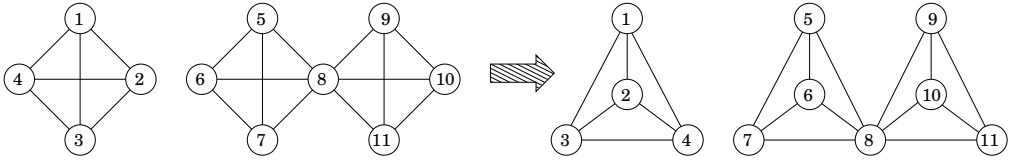
Übung 51. *Ist der Peterson-Graph planar?*



Aus den Sätzen von Wagner bzw. Kuratowski ergibt sich kein effizienter Planaritätstest.

Satz: Ein Graph ist genau dann planar, wenn jede seiner 2-fach Zusammenhangskomponenten planar ist.

- Jede Zusammenhangskomponente kann separat und unabhängig von den anderen auf Planarität getestet und eingebettet werden.
- Durch „Schnittpunkte miteinander verbundene Zusammenhangskomponenten“ können einzeln eingebettet werden.



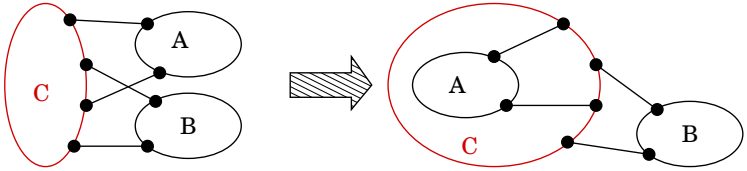
Es reicht also aus, Planarität von 2-fach Zusammenhangskomponenten zu testen, denn die haben keine Schnittpunkte.

Erkennen von planaren Graphen

Für einen 2-fach zusammenhängenden Graphen gilt: Jeder Knoten liegt auf einem Kreis. Wird der Knoten entfernt, so existiert noch ein anderer Weg zu diesem Knoten.

Algorithmus von Auslander und Parter (1961), korrigiert von Goldstein (1963)

- Bestimme einen Kreis C im 2-fach zusammenhängenden Graphen G.
- Entferne den Kreis aus G und zerlege die verbleibenden Teile des Graphen rekursiv.
- Füge die Einbettungen der Teile mittels C zu einer Einbettung von G zusammen. Überschneiden sich zwei Segmente, so platziere eins innerhalb und eins außerhalb von C. Sonst können beide innerhalb oder beide außerhalb platziert werden.



- Überschneidet sich ein Segment mit zwei bereits eingebetteten Segmenten, wobei eins innerhalb und eins außerhalb von C liegt, dann ist keine Einbettung möglich.

Knotenfärbung planarer Graphen

Satz: Jeder planare Graph kann mit sechs Farben gefärbt werden.

Beweis: Es gibt einen Knoten v mit Knotengrad höchstens 5.

- Wir berechnen eine Färbung für $G - \{v\}$ mit 6 Farben.
- Anschließend fügen wir v mit den dazu gehörenden Kanten wieder ein.

Da der Knoten v in G höchstens 5 Nachbarn hat, ist noch eine Farbe für v frei. □

Satz: Jeder planare Graph kann mit fünf Farben gefärbt werden.

Beweis: Betrachte eine Einbettung von G in die Ebene.

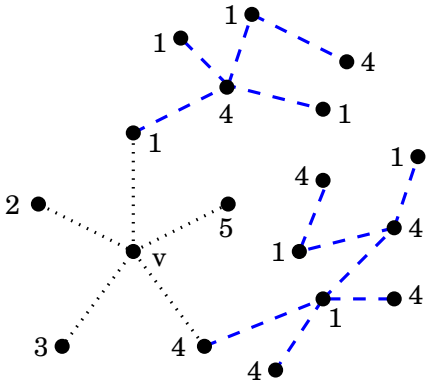
Fall 1: Wenn ein Knoten v mit Knotengrad höchstens vier in G existiert, dann färben wir $G - \{v\}$ mit 5 Farben und fügen Knoten v mit den dazu gehörenden Kanten wieder ein.

Da Knoten v nur maximal vier Nachbarn hat, ist noch eine Farbe für Knoten v frei.

Fall 2: Alle Knoten haben mindestens den Grad fünf. Sei v ein Knoten mit Grad fünf. Auch hier berechnen wir zunächst eine Färbung von $G - \{v\}$ mit 5 Farben.

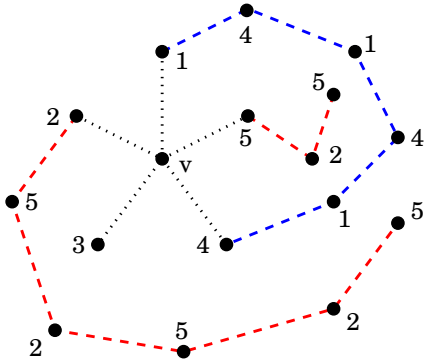
Falls die Nachbarn von v mit nur vier verschiedenen Farben gefärbt sind, kann v mit den dazu gehörenden Kanten eingefügt und mit der verbleibenden freien Farbe gefärbt werden.

Seien nun also die Nachbarn $w_1, \ldots, w_5$ von v mit fünf verschiedenen Farben gefärbt, dabei sei w_i mit Farbe i gefärbt.



Falls w_1 und w_4 in dem durch die Knoten mit Farben 1 und 4 induzierten Teilgraphen H in verschiedenen Komponenten H_1 und H_4 liegen:
Vertausche in H_1 die Farben 1 und 4. Das ist zulässig, weil die Knoten nur benachbart sind zu Knoten mit den Farben 2, 3 oder 5.
Färbe dann den Knoten v mit Farbe 1 und füge v mit den dazu gehörenden Kanten wieder ein.

Seien wieder die Nachbarn $w_1, \ldots, w_5$ von v mit fünf verschiedenen Farben gefärbt, dabei sei w_i mit Farbe i gefärbt.



Falls w_1 und w_4 in H verbunden sind, betrachte den durch die Knoten mit Farben 2 und 5 induzierten Teilgraphen H'. Die Knoten w_2 und w_5 können in H' nicht verbunden sein.
Vertausche die Farben 2 und 5 in der Komponente von w_2 in H'. Färbe v mit Farbe 2 und füge v mit den dazu gehörenden Kanten wieder ein. □

Knotenfärbung planarer Graphen

Satz: Jeder planare Graph kann mit vier Farben gefärbt werden.

- Dieser Satz wurde 1852 von *Augustus de Morgan* formuliert.
- *Alfred Kempe* liefert 1879 den ersten „Beweis“.
- 1890 fand *Percy Heawood* einen Fehler in Kempes Beweis.
- 1977 wurde der Satz von *Appel* und *Haken* bewiesen.
  - riesige Anzahl von Fallunterscheidungen nötig
  - Fallunterscheidungen wurden durch Computer gelöst
  - daher bei einigen Mathematikern umstritten
- 1995 geben *Robertson*, *Sanders*, *Seymour* und *Thomas* einen wesentlich kürzeren Beweis, allerdings werden immer noch Computer genutzt.

Aber: Es ist auch für planare Graphen NP-vollständig, zu entscheiden, ob drei Farben zur Knotenfärbung ausreichen.

3-Knotenfärbung für planare Graphen ist NP-vollständig

3-Knotenfärbung für planare Graphen ist in NP:

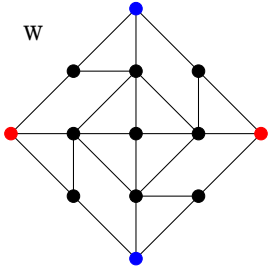
- Bestimme für den gegebenen Graphen $G = (V, E)$ nicht-deterministisch eine Zuordnung $f : V \to \{1, 2, 3\}$ von Farben zu Knoten.
- Prüfe für jede Kante $\{u, v\} \in E$, ob $f(u) \neq f(v)$ gilt.

3-Knotenfärbung $\leq_p$ 3-Planar-Knotenfärbung

Zunächst benötigen wir eine Einbettung des Graphen G in die Ebene, und zwar so, dass sich in einem Punkt höchstens zwei Kanten schneiden.

Eine solche Kreuzung zweier Kanten wird durch eine Kopie des rechts stehenden Graphen W ersetzt.

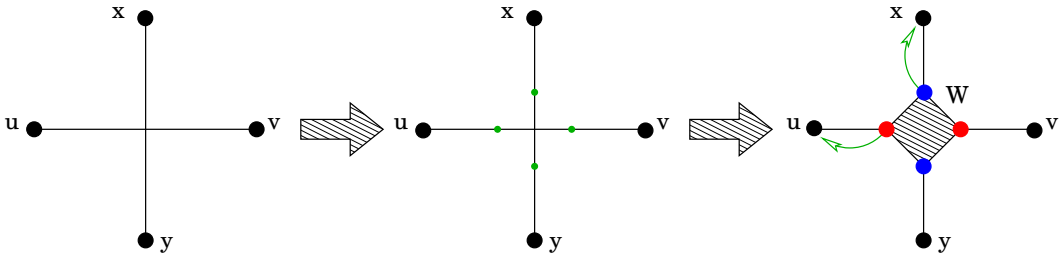
Die gefärbten „Eckknoten“ sind von zentraler Bedeutung.



3-Knotenfärbung für planare Graphen ist NP-vollständig

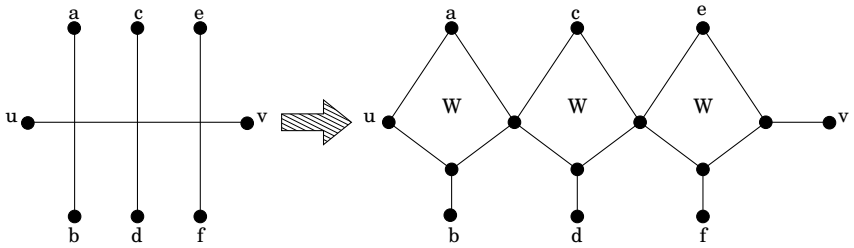
Die Ersetzung einer Kreuzung der Kanten $\{u, v\}$ und $\{x, y\}$ erfolgt in drei Schritten:

- Zunächst werden die beiden Kanten, die sich kreuzen, aufgeteilt.
- Dann wird eine Kopie des Graphen W eingefügt.
- Schließlich werden u und x mit je einem Knoten aus W verschmolzen.



3-Knotenfärbung für planare Graphen ist NP-vollständig

Wird eine Kante von mehreren anderen Kanten geschnitten, so müssen die Kopien von W wie folgt verkettet werden:

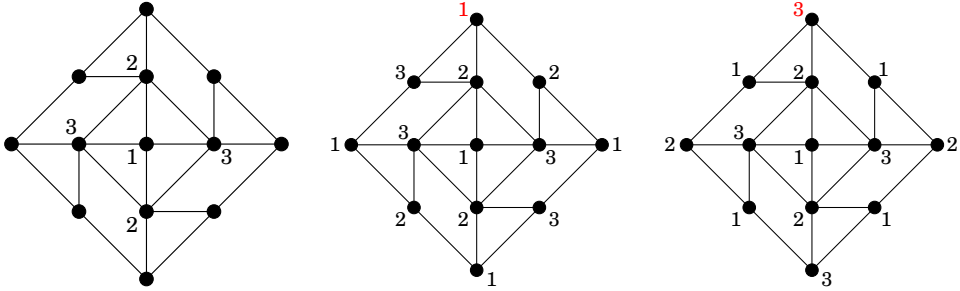


Der Graph W ist planar, 3-färbbar und hat folgende Eigenschaften:

- Bei jeder Färbung haben gegenüberliegende Eckknoten dieselbe Farbe.
- Eine Färbung gegenüberliegender Eckknoten mit derselben Farbe kann zu einer Färbung aller Knoten von W erweitert werden.

3-Knotenfärbung für planare Graphen ist NP-vollständig

Um die obige Aussage zu verstehen, färben wir zunächst den mittleren Knoten mit einer Farbe. Die umliegenden Knoten müssen dann abwechselnd mit den anderen zwei Farben gefärbt werden, siehe linke Abbildung.

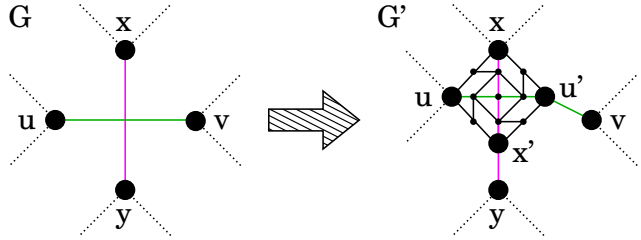


Der obere Knoten kann dann mit einer von zwei Farben gefärbt werden. Die Färbung der verbleibenden Knoten ist damit festgelegt: im mittleren Bild im Uhrzeigersinn, im rechten Bild entgegen dem Uhrzeigersinn.

Andere Färbungen ergeben sich, wenn der mittlere Knoten anders gefärbt wird oder seine umliegenden Knoten.

3-Knotenfärbung für planare Graphen ist NP-vollständig

Der aus Graph G durch obige Transformation entstehende Graph G' ist planar, in polynomieller Zeit berechenbar und es gilt: G ist 3-färbbar $\iff$ G' ist 3-färbbar



$\Rightarrow$ G ist 3-färbbar
Es existiert eine Färbung f' für G', sodass $f(u) = f'(u) = f'(u')$ und $f(x) = f'(x) = f'(x')$ gilt, und $f(v) = f'(v)$ und $f(y) = f'(y)$.

$\Leftarrow$ G' ist 3-färbbar
Für jede Färbung f' gilt $f'(u) = f'(u')$ und $f'(x) = f'(x')$, daher ist f' eingeschränkt auf die Knotenmenge V eine Färbung von G.

Für planare Graphen gilt: Maximum-Clique ist in P.

- Da ein planarer Graph G keinen K_5 enthalten kann, gilt $\omega(G) \leq 4$ und daher können alle 4-elementigen Teilmengen aufgezählt und getestet werden.
- Laufzeit: $\mathcal{O}(n^4 \cdot |G|)$

Independent-Set: NP-complete even when the maximum degree is at most 6.[16]

Minimum-Clique-Cover: NP-complete on $(C_4, C_5, K_4, \mathit{diamond})$-free $\cap$ planar.[17]

Weitere Graphklassen und Komplexitäten verschiedener Probleme finden sich auf der Web-Seite `https://www.graphclasses.org/`.

[16] M.R. Garey, D.S. Johnson, L. Stockmeyer: Some simplified NP-complete graph problems. Theor. Comp. Sci. (1), 1976, 237–267

[17] D. Kral, J. Kratochvil, Z. Tuza, G.J. Woeginger: Complexity of coloring graphs without forbidden induced subgraphs. Proceedings of the 27th International Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 2204, 254-262 (2001)

Minoren werden heute auch zum Lösen von Optimierungsproblemen mittels Quanten-Annealer[18] benötigt, um das logische auf das physikalische Modell abzubilden.

- Qubits (kurz für Quantum-Bits) können die Werte 0 und 1 annehmen, aber auch alle anderen Werte dazwischen.
- Lösung kombinatorischer Optimierungsprobleme ähnlich Simulated-Annealing.

Im Folgenden sind $\hat{\sigma}_x^{(i)}$ und $\hat{\sigma}_z^{(i)}$ Pauli-Matrizen, die auf Qubit q_i wirken, h_i und $J_{i,j}$ sind Qubit-Bias bzw. Stärke der Kopplung zwischen zwei Qubits.

$$H(s) = A(s) \cdot \sum_i \hat{\sigma}_x^{(i)} + B(s) \cdot \left(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)} \right)$$

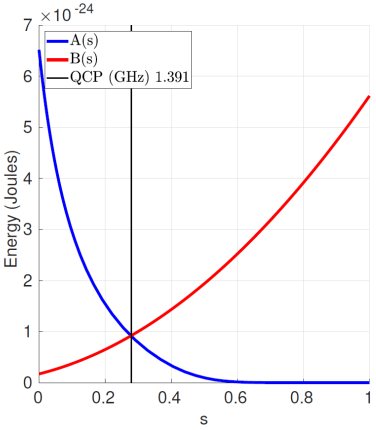
Der initiale (linker Teil) Hamiltonian beschreibt den Grundzustand des Systems, der vom Nutzer nicht angegeben werden muss. Die zu lösende Aufgabe wird durch den finalen (rechter Teil) Hamiltonian beschrieben.

[18] `https://docs.dwavesys.com/docs/latest/c_gs_2.html`

The lowest-energy state of the initial Hamiltonian is when all qubits are in a superposition[19] state of 0 and 1.

Annealing begins at $s = 0$ with $A(0) \gg B(0)$ and ends at $s = 1$ with $A(1) \ll B(1)$. As it anneals, it introduces the problem Hamiltonian, and it reduces the influence of the initial Hamiltonian.

The lowest-energy state of the final Hamiltonian is the answer to the problem that you are trying to solve. The final state is a classical state, and includes the qubit biases and the couplings between qubits.



[19] Überlagerungszustand, in denen ein Gegenstand sich in zwei Zuständen gleichzeitig befindet.

The minima of objective functions can be formulated in two forms, QUBO-formulation

$$\min_{\mathbf{X}} E(\mathbf{X}) = \min_{\mathbf{X}} \left(\sum_{i=1}^{n} Q_{i,i} x_i + \sum_{i<j} Q_{ij} x_i x_j \right) \qquad x_i \in \{0, 1\}$$

and Ising-formulation

$$\min_{\mathbf{S}} E(\mathbf{S}) = \min_{\mathbf{S}} \left(\sum_{i=1}^{n} h_i s_i + \sum_{i<j} J_{ij} s_i s_j \right) \qquad s_i \in \{-1, +1\}$$

The two formulations are equivalent via bijective relations $h_i = 1/2(Q_{i,i} + \sum_j Q_{ij})$, and $J_{ij} = Q_{ij}/4$.

aus: P. Ueberholz, C. Gebler, J. Rethmann. QUBO Models for the FIFO Stack-Up Problem and Experimental Evaluation on a Quantum Annealer. Springer Nature Computer Science, 2024.

QUBO: Quadratic unconstrained binary optimization[20]

- QUBO is an NP hard problem, and for many classical problems from theoretical computer science, like maximum cut, graph coloring and the partition problem, embeddings into QUBO have been formulated.
- Due to its close connection to Ising models, QUBO constitutes a central problem class for adiabatic quantum computation, where it is solved through a physical process called quantum annealing.
- QUBO can be solved using integer linear programming solvers. This is possible since QUBO can be reformulated as a linear constrained binary optimization problem. To achieve this, substitute the product $x_i x_j$ by an additional binary variable $z_{ij} \in \{0, 1\}$ and add the constraints $x_i \geq z_{ij}$, $x_j \geq z_{ij}$ and $x_i + x_j - 1 \leq z_{ij}$.

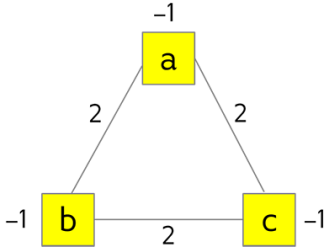
[20] https://en.wikipedia.org/wiki/Quadratic_unconstrained_binary_optimization

Satisfiability[21] exactly one is true: $a + b + c = 1$

$E(a, b, c) = (a + b + c - 1)^2 = 2ab + 2ac + 2bc - a - b - c + 1$ nimmt einen minimalen Wert ein, genau dann wenn entweder a oder b oder c eins ist.

Dies kann durch den nebenstehenden Graphen dargestellt werden.

Im logischen Modell lässt sich jedes Qubit mit jedem anderen Qubit verschalten, es ist also prinzipiell ein vollständiger Graph möglich.



Quanten-Annealer werden von der kanadischen Firma D-Wave verkauft oder auch Rechenzeit vermietet. Zwei Modelle mit 2048 bzw. 5520 Qubits und fester Topologie Chimera bzw. Pegasus.

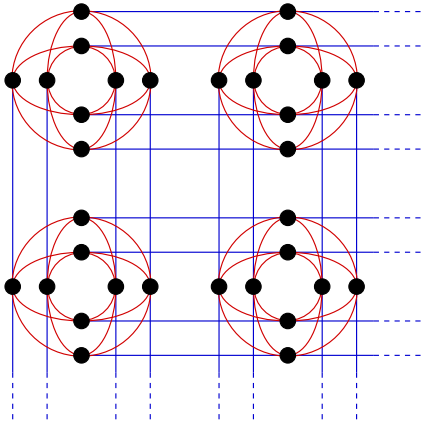
[21] `https://docs.dwavesys.com/docs/latest/c_gs_6.html`

Chimera-Topologie[22]: Jedes Qubit

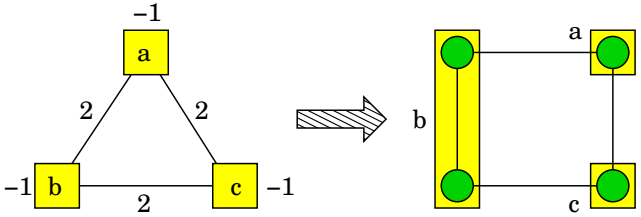
- ist mit vier anderen Qubits in seiner Zelle
- und mit maximal zwei weiteren Qubits aus anderen Zellen verbunden.

Das obige logische Modell muss mittels Minor-Embedding auf das physikalische Modell abgebildet werden, weil die Chimera-Topologie kein Dreieck enthält.

Es ist möglich, physikalische Qubits zu koppeln, sodass sie gemeinsam ein logisches Qubit repräsentieren.



[22] `https://docs.dwavesys.com/docs/latest/c_gs_4.html`



Um solche Ketten zu formen, müssen die Kopplungsparameter entsprechend angepasst werden, was wiederum Änderungen an anderen Stellen des Hamiltonian erfordert[23].

Problem: Physische Qubits sind nicht perfekt, daher brechen solche Ketten in der Praxis. Je länger die Ketten umso wahrscheinlicher ist ein Bruch.

Es werden also Minor-Embeddings gesucht, die möglichst kurze Ketten haben. Erfolgt bei D-Wave heuristisch[24].

[23] i'X Spezial 2021 - Quantencomputer

[24] `https://docs.ocean.dwavesys.com/en/stable/docs_minorminer/source/intro.html`

[25] `https://docs.dwavesys.com/docs/latest/c_gs_7.html`

Übersicht

Die Algorithmen von

- *Dijkstra* zur Berechnung *kürzester Wege* sowie
- *Prim* zur Berechnung *minimaler Spannbäume*

verwenden eine Datenstruktur zum Speichern von Knoten mit Operationen `ExtractMin` und `DecreaseKey`.

Wir kennen drei mögliche Implementierungen:

	Array	Linked List	Binary Heap
`ExtractMin`	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
`DecreaseKey`	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

`DecreaseKey` setzt voraus, dass jeweils ein Zeiger auf jedes Element vorhanden ist, um das Element effizient zu finden, um also keine Zeit mit Suchen zu verschwenden.

Zugriffsmethoden: (komplett)

	Linked List	Binary Heap
`MakeHeap`	$\Theta(1)$	$\Theta(1)$
`Insert`	$\Theta(1)$	$\Theta(\log n)$
`Minimum`	$\Theta(n)$	$\Theta(1)$
`ExtractMin`	$\Theta(n)$	$\Theta(\log n)$
`Union`	$\Theta(1)$	$\Theta(n)$
`DecreaseKey`	$\Theta(1)$	$\Theta(\log n)$
`Delete`	$\Theta(1)$	$\Theta(\log n)$

- Auch bei `Delete` muss das Element direkt zugreifbar sein.
- Lassen sich Vorteile der verketteten Listen und Heaps vereinigen?
- Können die rot markierten Laufzeiten verbessert werden?

Folgende Operationen muss eine Priority Queue unterstützen:

- `MakeHeap()` erzeugt einen neuen Heap ohne Elemente.
- `Insert(`H, x`)` fügt Knoten x in Heap H ein.
- `Minimum(`H`)` liefert einen Zeiger auf den Knoten mit minimalem Element im Heap H.
- `ExtractMin(`H`)` entfernt das minimale Element aus Heap H und liefert einen Zeiger auf den Knoten.
- `Union(`H_1, H_2`)` erzeugt einen neuen Heap, der die Elemente aus H_1 und H_2 enthält.
- `DecreaseKey(`H, x, k`)`$^{(*)}$ weist dem Knoten x im Heap H einen neuen, kleineren Wert k zu.
- `Delete(`H, x`)`$^{(*)}$ entfernt Knoten x aus Heap H.

$^{(*)}$ Der Zeiger auf x muss bekannt sein, um Suchen zu vermeiden.

Übersicht

Ein Baum mit $N+1$ Blättern und N inneren Knoten ist balanciert, wenn jedes Blatt eine Tiefe aus $\mathcal{O}(\log(N))$ hat.

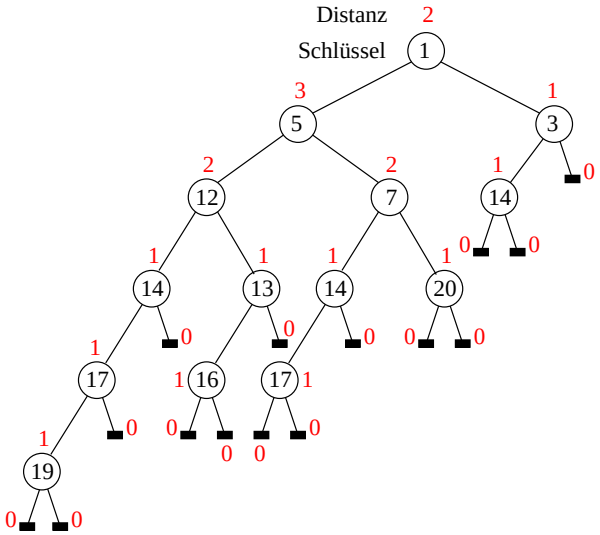
Implementierung von Priority-Queues: Es reicht eine wesentlich schwächere Forderung, um obige Operationen in der vorgegebenen Zeit auszuführen.

Linksbäume (leftist trees) sind binäre, heap-geordnete, links-rechts-geordnete Bäume, die in ihren inneren Knoten Elemente mit Schlüsseln[26] speichern. Die Heap-Ordnung bezieht sich auf die Schlüssel, nicht auf die eigentlichen Objekte.

In unseren Abbildungen sind die Blätter oft nicht gezeichnet.

[26] Schlüssel bezeichnet hier kein eindeutiges Kriterium sondern bspw. die Priorität eines Eintrags. Schlüsselwerte können also mehrfach vorhanden sein.

Beispiel:


Distanz
Schlüssel
2
1
3
5
1
3
0
2
12
2
7
1
14
0
0
1
14
1
13
1
14
1
20
0
0
0
0
0
0
1
17
1
16
17
1
0
0
0
0
0
0
1
19
0
0

Eigenschaften:

- Der Schlüsselwert eines Knotens ist immer kleiner gleich den Schlüsselwerten aller seiner Kinder: Min-Heap!
- Jeder Knoten besitzt einen Distanzwert.
  - Blätter: Der Distanzwert ist 0.
  - Innere Knoten: Distanzwert des rechten Kindes plus 1.
  - Distanzwert rechtes Kind $\leq$ Distanzwert linkes Kind.
- Aufgrund der letzten Bedingung hat der Linksbaum seinen Namen: Der Baum „hängt“ nach links, er ist links „tiefer“ als rechts.

Übung 52. *Kann in einem Linksbaum der rechte Teilbaum eines Knotens eine größere Tiefe als der linke Teilbaum haben?*

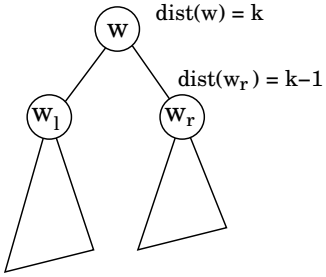
Wir werden zeigen: Sei w ein Knoten mit $dist(w) = k$. Dann enthält der Teilbaum mit Wurzel w mindestens 2^k Knoten. $\rightarrow$ In einem Linksbaum hat das rechteste Blatt eine Tiefe von höchstens $\mathcal{O}(\log(N))$.

Lemma: Sei w ein Knoten mit $dist(w) = k$. Dann enthält der Teilbaum mit Wurzel w mindestens 2^k Knoten.

Beweis mittels vollständiger Induktion:

I.A. Für $dist(w) = 0$ gilt: w ist ein Blatt und der Teilbaum mit Wurzel w besteht aus $2^0 = 1$ Knoten. ✓

I.S. Der Knoten w habe den linken Nachfolger w_l und den rechten Nachfolger w_r.



Dann gilt $k = dist(w) = dist(w_r)+1$ und $dist(w_l) \geq dist(w_r)$ nach Definition.

Damit haben die Teilbäume mit den Wurzeln w_l und w_r nach I.V. mindestens 2^{k-1} Knoten.

Der Teilbaum mit Wurzel w enthält also mindestens $2^{k-1} + 2^{k-1} = 2^k$ Knoten. ✓

MakeHeap(): Erzeuge ein Blatt mit Distanzwert 0.
$\rightarrow$ Laufzeit: $\mathcal{O}(1)$

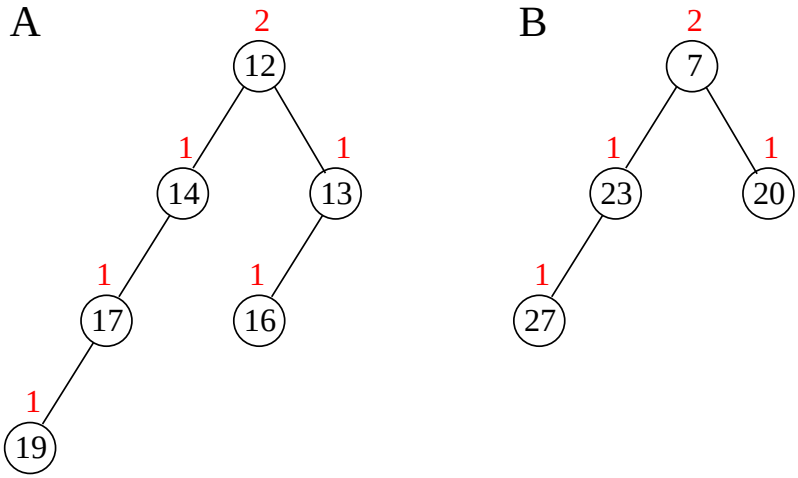
Minimum(D): Gib das Wurzel-Element zurück.
$\rightarrow$ Laufzeit: $\mathcal{O}(1)$

Alle anderen Operationen lassen sich auf das Verschmelzen von zwei Linksbäumen zurückführen!

UNION(D_1, D_2): rekursiv

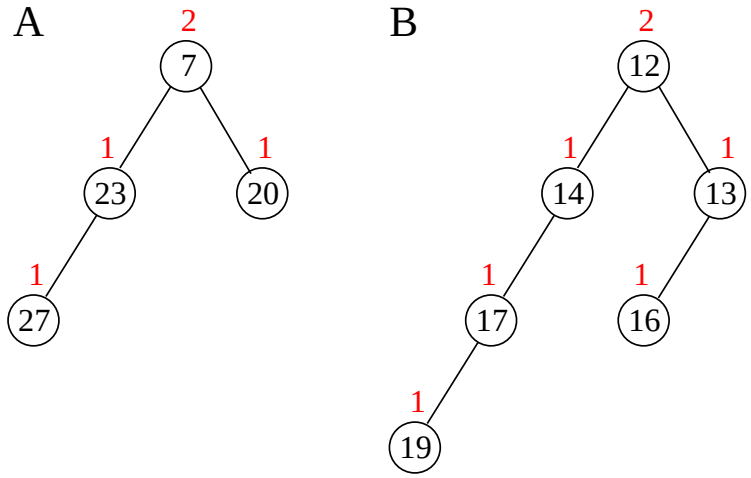
- Rekursionsende: Wenn D_1 oder D_2 ein Blatt ist, dann ist das Ergebnis der Linksbaum D_2 bzw. D_1.
- o.B.d.A: Der Schlüssel an Wurzel von D_1 ist kleiner als der Schlüssel an Wurzel von D_2 – sonst tausche Linksbäume.
  Dies ist notwendig, damit nach dem Verschmelzen die Heap-Bedingung gilt.
  (a) UNION(D_1.RECHTS, D_2)
  (b) Ist die Distanz vom (neuen) rechten Teilbaum von D_1 größer als die Distanz vom linken Teilbaum von D_1, dann werden die Teilbäume von D_1 vertauscht.
  Dies ist notwendig, damit die Distanzbedingung für die beiden Teilbäume gilt.

UNION(A,B): Initial



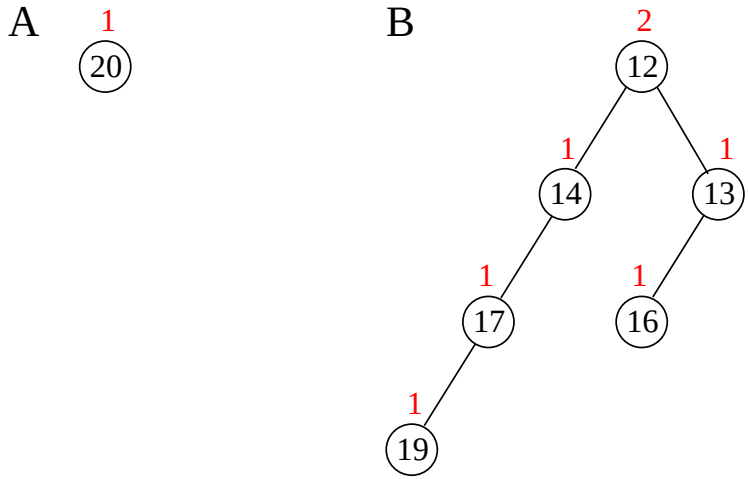
→ Bäume vertauschen

UNION(A,B): Bäume vertauscht



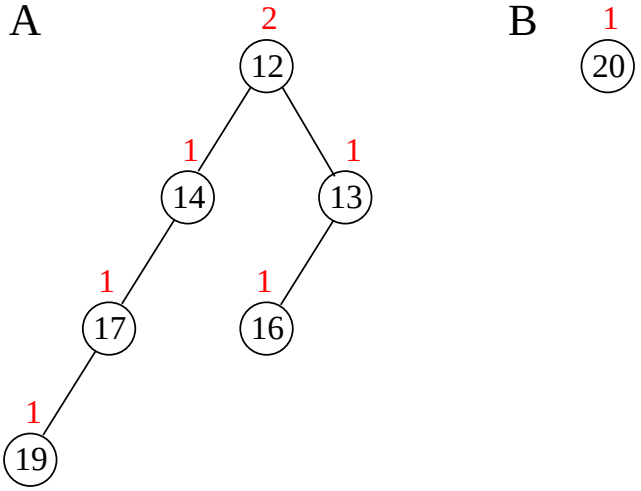
→ verschmelzen von A.rechts und B

UNION(A,B): verschmelzen von A und B



→ Bäume vertauschen

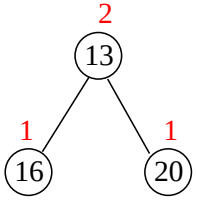
UNION(A,B): Bäume vertauscht



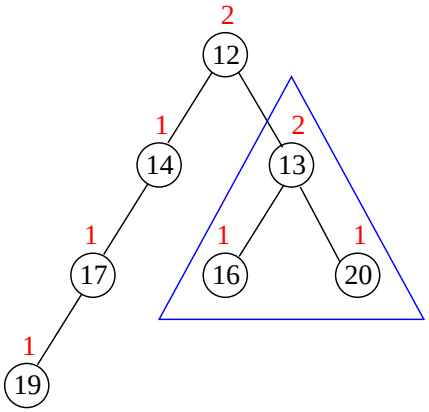
→ verschmelzen von A.rechts und B

UNION(A,B): verschmelzen von A und B

jetzt besteht A.rechts nur noch aus einem Blatt
→ Ergebnis der letzten Rekursion:

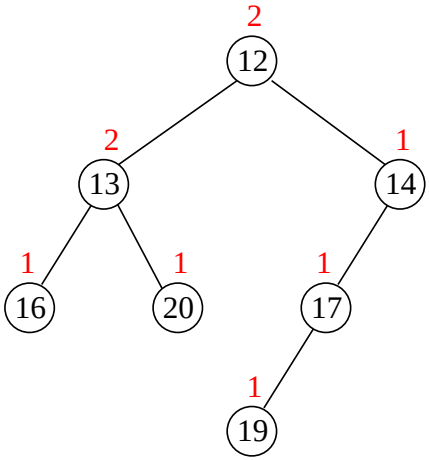


UNION(A,B): vorläufiges Ergebnis der zweiten Rekursion



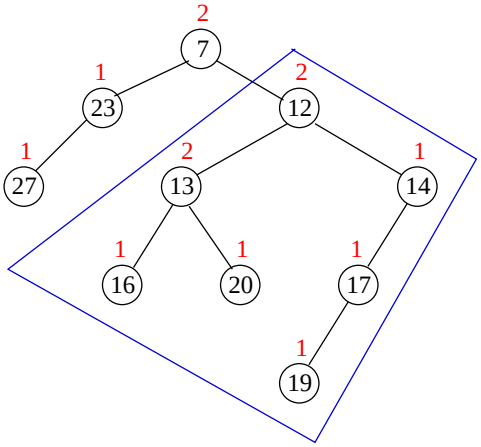
Distanzwert rechts (neu) größer als Distanzwert links → vertausche die beiden Teilbäume

UNION(A,B): Ergebnis der zweiten Rekursion



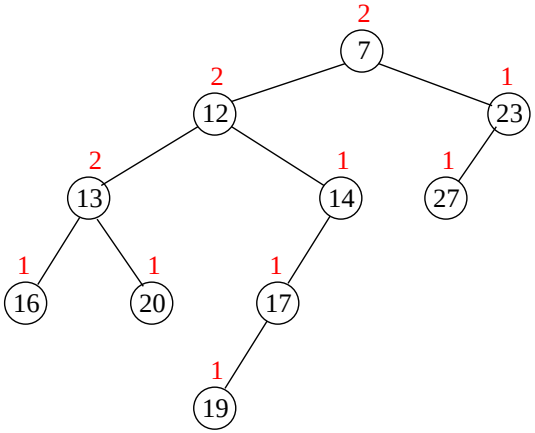
Anmerkung: Auch hier ist der rechte Teilbaum der Wurzel tiefer als der linke Teilbaum.

UNION(A,B): vorläufiges Ergebnis der ersten Rekursion



Distanzwert rechts (neu) größer als Distanzwert links → vertausche die beiden Teilbäume

UNION(A,B): Ergebnis der ersten Rekursion



Laufzeit: Ist beschränkt durch die Summe der Längen der beiden Pfade von der Wurzel zum jeweils rechtesten Blatt. $\rightarrow \mathcal{O}(\log(N))$

INSERT(D, x):

- Erzeuge einen neuen Linksbaum E, der nur einen einzigen inneren Knoten x mit Distanz 1 hat.
- UNION(D, E)

$\rightarrow$ Laufzeit: $\mathcal{O}(\log(N))$

EXTRACTMIN(D):

- Entferne die Wurzel.
- Verschmelze die beiden Teilbäume der Wurzel.

$\rightarrow$ Laufzeit: $\mathcal{O}(\log(N))$

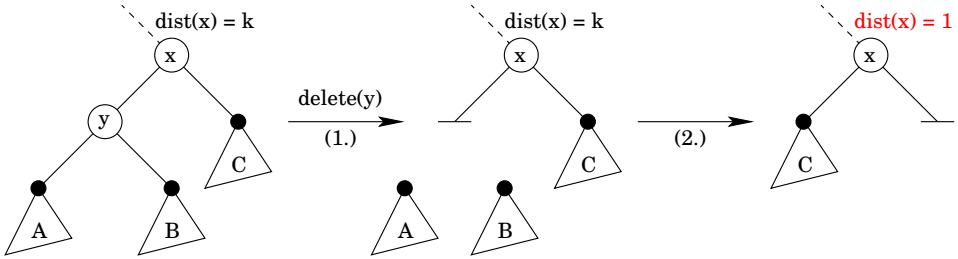
DELETE(D, x):

- Der Linksbaum zerfällt beim Entfernen von x in den oberhalb von x liegenden Teilbaum und in den linken und rechten Teilbaum.
- Ersetze den Knoten x durch ein Blatt b.
- Tausche ggf. b mit seinem Geschwister, um links den Teilbaum mit größerer Distanz anzuordnen.
- Adjustiere die Distanzwerte von b bis zur Wurzel.
- Verschmelze die drei Teilbäume miteinander.

$\rightarrow$ Laufzeit: $\mathcal{O}(\log(N))$

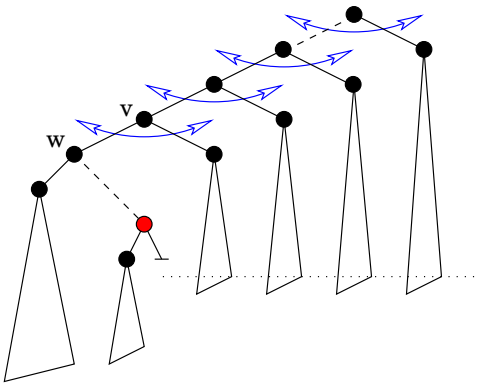
Warum? Das Adjustieren der Distanzen und das Vertauschen von Teilbäumen auf dem Pfad zur Wurzel ist doch proportional zur Höhe des Baumes, im schlimmsten Fall also $\Theta(N)$, oder?

Beispiel: Im folgenden Ausschnitt eines Linksbaums wird Knoten y gelöscht.



- Zunächst werden die Teilbäume A und B vom zu löschenden Knoten abgetrennt. Diese werden später mittels UNION dem Linksbaum wieder hinzugefügt.
- Da der Vorgänger x einen nicht-leeren, rechten Teilbaum C hat, müssen der linke und rechte Nachfolger von x getauscht und die Distanz von x korrigiert werden.
- Korrigiere Vorgängerdistanzen, falls x rechter Nachfolger ist.

Wenn beim Korrigieren der Vorgängerdistanzen ein Knoten w erreicht wird, der linker Nachfolger seines Vorgängers v ist, müssen ggf. Teilbäume getauscht werden.



Dies erfolgt aber nur, wenn der rechte Teilbaum von v tiefer ist als der linke. Setzt sich das Tauschen bis zur Wurzel fort, ist der Baum bis w im Wesentlichen balanciert und hat nur logarithmische Tiefe.

Das Adjustieren der Distanzwerte benötigt höchstens $\log(N)$ viele Schritte, denn die Distanzwerte beginnen bei 0 mit dem erzeugten Blatt und werden bei der Adjustierung auf dem Weg zur Wurzel immer schrittweise um genau 1 größer.

Wir hatten bereits gezeigt, dass der Weg von der Wurzel zum rechtesten Blatt höchstens die Länge $\log(N)$ hat. $\rightarrow$ Das Adjustieren der Distanzwerte und das Tauschen der Teilbäume hat ebenfalls eine logarithmische Laufzeit.

DecreaseKey(D, x, k):

- Delete(D, x)
- Ändere den Schlüssel von x auf k.
- Insert(D, x)

$\rightarrow$ Laufzeit: $\mathcal{O}(\log(N))$

Zusammenfassung

	Linked List	Binary Heap	Linksbaum
`MakeHeap`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`Insert`	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
`Minimum`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`ExtractMin`	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
`Union`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`DecreaseKey`	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
`Delete`	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Übung 53.

- *Fügen Sie die Werte 1,2,3,4,5,6,7 in dieser Reihenfolge in einen initial leeren Linksbaum ein.*
- *Führen Sie anschließend* ExtractMin() *und danach* DecreaseKey(6, 1) *aus.*
- *Geben Sie nach jedem Schritt den resultierenden Linksbaum an.*
- *Kann ein Linksbaum zu einer linearen Liste entarten? Begründen Sie Ihre Antwort.*

Übersicht

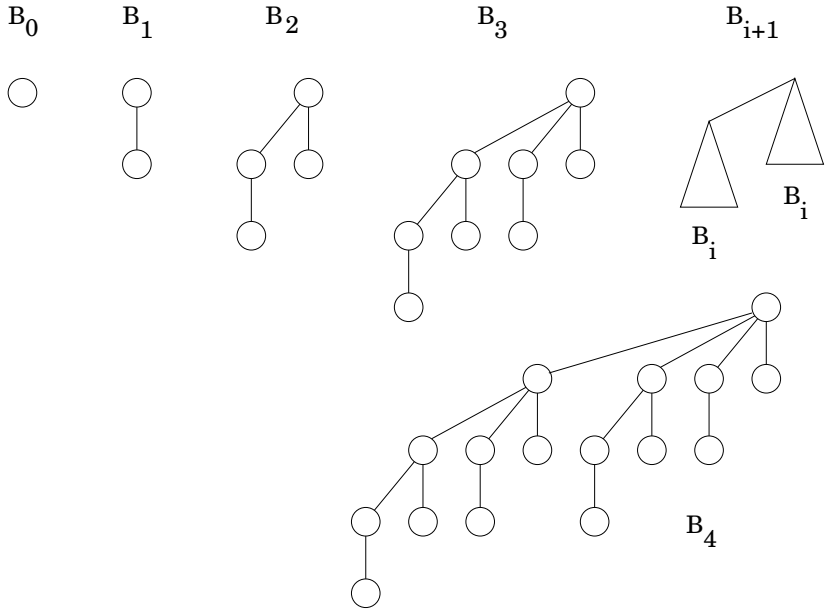
Binomialbäume sind heap-geordnete Bäume, die in allen Knoten Elemente mit Schlüsseln speichern. (Min-Heap)

Ein Binomialbaum vom Typ

- B_0 besteht aus genau einem Knoten.
- B_{i+1}, $i \geq 0$, besteht aus zwei Kopien der Binomialbäume vom Typ B_i, indem man die Wurzel der einen Kopie zum Kind der Wurzel der anderen Kopie macht.

Ihren Namen verdanken die Binomial-Queues B_k der Tatsache, dass auf der Ebene i die Anzahl der Knoten gerade $\binom{k}{i}$ beträgt.

Binomial Heap



Eigenschaften eines Binomialbaums vom Typ B_k:

1. B_k hat 2^k Knoten und Höhe k.
2. In Ebene i hat B_k genau $\binom{k}{i}$ viele Knoten.
3. Teilbäume der Wurzel sind vom Typ $B_{k-1}, B_{k-2}, \ldots, B_0$.

Beweis: Induktion nach k. Für $k = 0$ gelten die Aussagen.

1. B_k entsteht, indem ein B_{k-1} unter einen anderen B_{k-1} gehängt wird, also gilt:
   - Anzahl Knoten: $2^{k-1} + 2^{k-1} = 2^k$
   - $depth(B_k) = depth(B_{k-1}) + 1 = (k-1) + 1 = k$.
2. Sei $D(k, i)$ die Anzahl der Knoten in Tiefe i von B_k. Da ein B_{k-1} unter einen anderen B_{k-1} gehängt wird, gilt:
   $D(k, i) = D(k-1, i) + D(k-1, i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$
3. siehe Bild auf vorheriger Folie
   - aus B_0 und B_0 entsteht B_1, also ein Teilbaum B_0
   - aus B_1 und B_1 entsteht B_2, also ein Teilbaum B_1 usw.

Speichern von n Elementen: Die Anzahl der benötigten Bäume ist gleich der Anzahl der Einsen in der Binärdarstellung von n. Sei $n = (d_{m-1} \dots d_0)$. Dann gilt:

$$\text{Baum vom Typ } B_j \text{ wird benötigt} \iff d_j = 1$$

Beispiel: Um $n = 13 = (1101)_2$ in einem Binomial-Heap zu speichern, werden Bäume vom Typ B_3, B_2 und B_0 benötigt.

Eine Menge von Bäumen nennt man in der Graphentheorie (und nicht nur da ☺) einen *Wald*.

Ein *Binomial-Heap vom Typ D_n* ist die Repräsentation einer Menge mit n Elementen durch Binomialbäume.

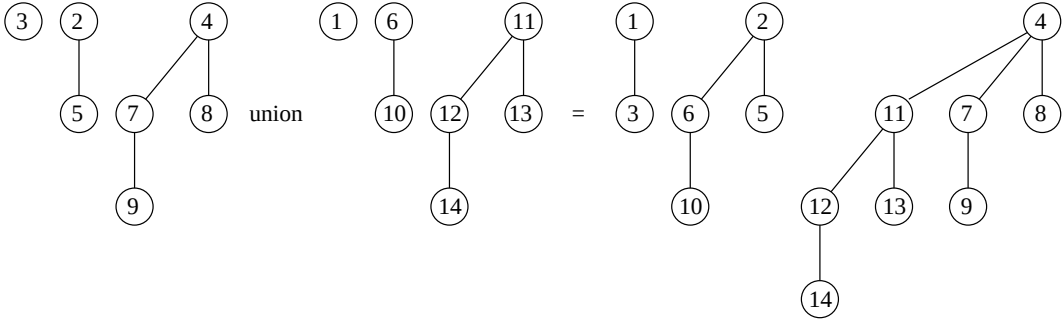
- Binomial-Heaps werden auch Binomial-Queues genannt.
- Die Anzahl der Bäume ist höchstens $\log(n)$.
- Die Wurzeln der Bäume werden in einer verketteten Liste gespeichert.

`MakeHeap()`: Erzeuge eine neue, leere Wurzelliste.
$\rightarrow$ Laufzeit: $\Theta(1)$

`Minimum(`H`)`: Durchsuche die Wurzelliste.
$\rightarrow$ Laufzeit: $\Theta(\log(n))$

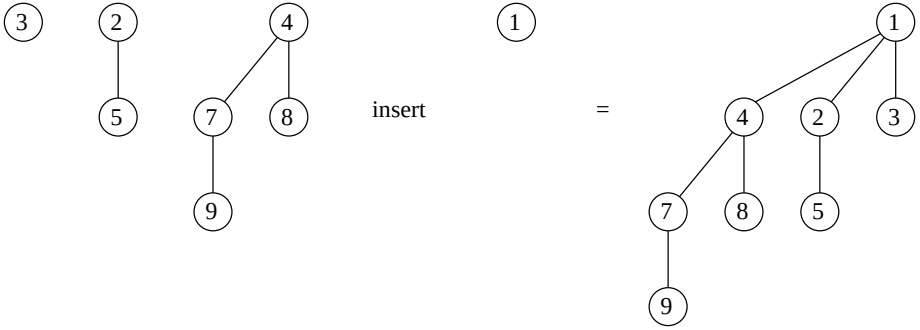
`Union(`H_1, H_2`):` Vereinige die Bäume aus beiden Binomial-Queues wie folgt:

- Zwei Bäume vom Typ B_0 ergeben einen Baum vom Typ B_1.
- Zwei Bäume vom Typ B_1 ergeben einen Baum vom Typ B_2.
- usw. analog zur Addition zweier Dualzahlen



$\rightarrow$ Laufzeit: $\Theta(\log(n+m))$

`Insert(`H, x`):` Vereinige H und den Baum vom Typ B_0, der den Knoten x enthält.

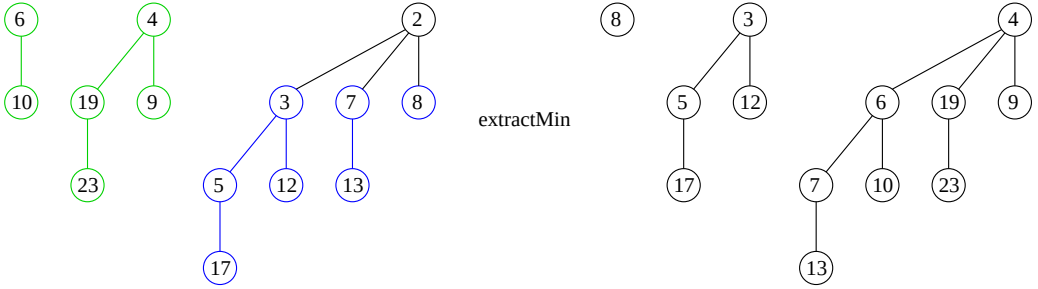


$\rightarrow$ Laufzeit: $\Theta(\log(n))$

`ExtractMin(H):`

- Sei B_j der Baum, der das minimale Element in seiner Wurzel speichert.
- Sei D_{n-2^j} der Wald ohne Baum B_j, unten grün markiert.
- Sei D_{2^j-1} der Wald, wenn im Baum B_j die Wurzel entfernt wird, unten blau markiert.

$\Rightarrow$ Vereinige D_{n-2^j} und D_{2^j-1}.



$\rightarrow$ Laufzeit: $\Theta(\log(n))$

`DecreaseKey(`H, x, k`):`

- Setze den Schlüssel von Knoten x auf den Wert k.
- `UpHeap(`H, x`)`: Laufe im Baum hoch und vertausche den Knoten mit seinem Vorgänger, falls der Vorgänger einen größeren Wert speichert.

$\rightarrow$ Laufzeit: $\Theta(\log(n))$

`Delete(`H, x`):`

- `DecreaseKey(`$H, x, -\infty$`)`
- `ExtractMin(`H`)`

$\rightarrow$ Laufzeit: $\Theta(\log(n))$

Zusammenfassung

Operation	Linked List	Binary Heap	Links-baum	Binomial Heap
MakeHeap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
DecreaseKey	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Delete	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Übung 54.

- *Fügen Sie die Werte 1,2,3,4,5,6,7 in dieser Reihenfolge in einen initial leeren Binomial Heap ein.*
- *Führen Sie anschließend* ExtractMin() *und danach* DecreaseKey(6, 1) *aus.*
- *Geben Sie nach jedem Schritt den resultierenden Binomial Heap an.*

Übersicht

Noch nicht zufriedenstellend beschleunigte Operationen:

- Insert
- DecreaseKey
- Delete

Idee der Fibonacci-Heaps:

- Verzichte beim Einfügen und Löschen auf die Vereinigung der Bäume.
- Hole dies erst bei ExtractMin nach.
- Vermerke nach jeder Operation das minimale Element.

Fibonacci-Heap

Fibonacci-Heap:

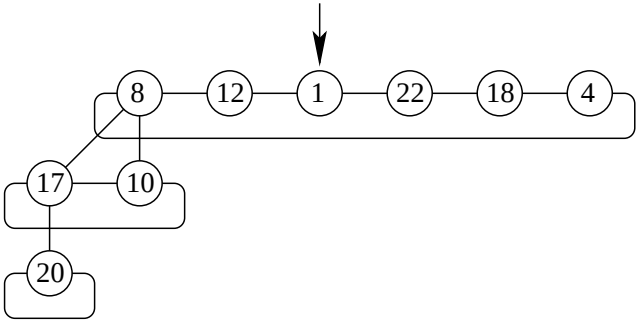
- Sammlung heap-geordneter Bäume. (Min-Heap)
- Struktur implizit durch erklärte Operationen definiert.

$\Rightarrow$ Jede mit den bereitgestellten Operationen aufbaubare Struktur ist ein Fibonacci-Heap.

Implementierung:

- Die Wurzeln der Bäume sind doppelt zyklisch verkettet.
- Zeiger auf das kleinste Element der Wurzelliste.
- Kinder der Knoten ebenfalls doppelt zyklisch verkettet.

Beispiel:



MakeHeap(): Erzeuge leeren Fibonacci-Heap: `NULL`-Zeiger
$\rightarrow$ Laufzeit: $\Theta(1)$

Minimum(H): Element, worauf Minimalzeiger zeigt.
$\rightarrow$ Laufzeit: $\Theta(1)$

UNION(H_1, H_2):

- Hänge die Wurzellisten von H_1 und H_2 aneinander.
- Setze neuen Minimalzeiger auf Minimum der Minimalelemente von H_1 und H_2.

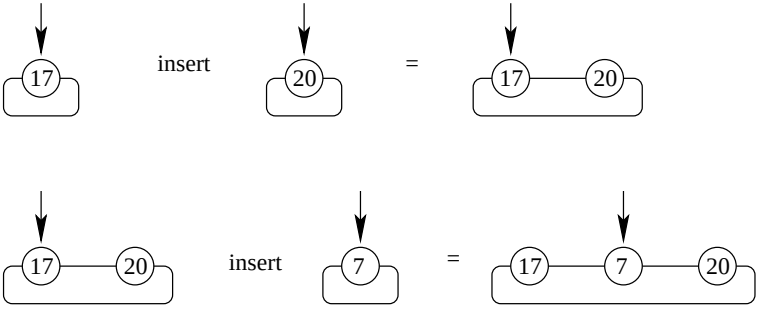
$\rightarrow$ Laufzeit: $\Theta(1)$

Weiteres Implementierungsdetail: Jeder Knoten hat einen *Rang* und ein *Markierungsfeld*.

- Der *Rang* entspricht der Anzahl der Kinder.
- Der Sinn des *Markierungsfeldes* wird bei DECREASEKEY erklärt.

INSERT(H, x):

- Erzeuge einen Fibonacci-Heap H', der nur x enthält.
- Der Rang von x ist 0, das Element ist nicht markiert.
- Vereinige H und H' mit der Operation UNION.



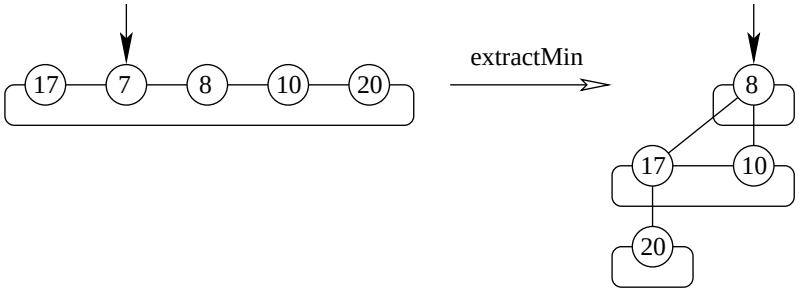
$\rightarrow$ Laufzeit: $\Theta(1)$

ExtractMin(H):

- Entferne den Minimalknoten u aus der Wurzelliste und bilde eine neue Wurzelliste durch Einhängen der Liste der Kinder von u an Stelle von u.
  $\rightarrow$ Laufzeit $\Theta(1)$, weil die Kinder zyklisch verkettet sind.
- Consolidate: Verschmelze nun solange zwei Bäume vom selben Rang, bis die Wurzelliste nur noch Bäume mit paarweise verschiedenem Rang enthält.
  - Beim Verschmelzen (kein Union!) 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 Kind von v' und das Markierungsfeld von v wird auf nicht markiert gesetzt.
  - Aktualisiere auch den Minimalzeiger.

Laufzeit: $\mathcal{O}(n)$ im worst-case, aber amortisiert nur $\mathcal{O}(\log(n))$.

Beispiel:

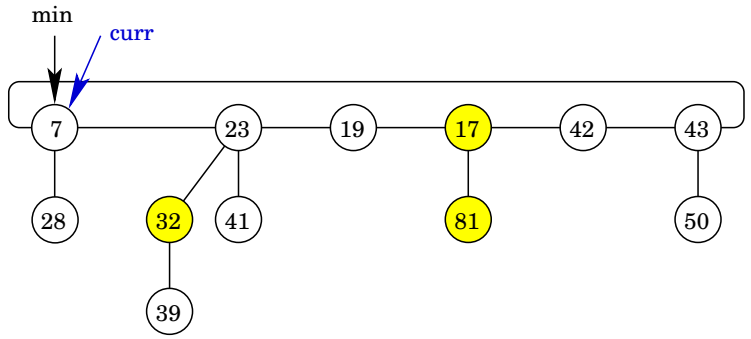


Effiziente Implementierung des Consolidate-Schrittes:

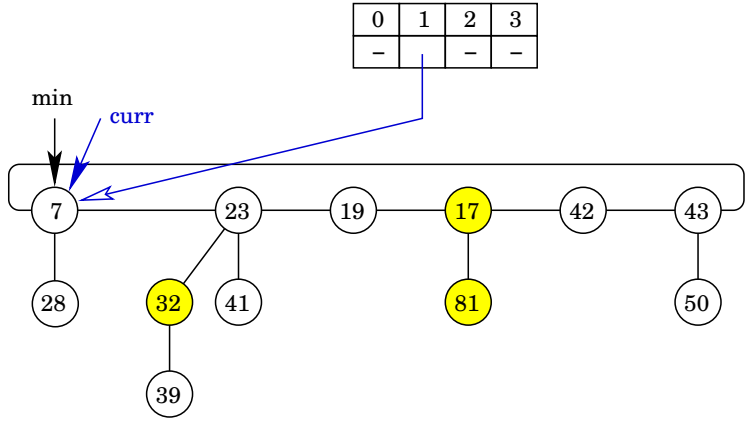
- verwende temporäres Array: Element `A[i]` zeigt auf einen Knoten x, so dass x die Wurzel eines Baums mit Rang `i` gilt.
- durchlaufe die Wurzelliste beginnend beim min-Zeiger:
  - sei `curr` der Zeiger auf den aktuell untersuchten Knoten
  - falls `A[rang(curr)] == NULL` ist, füge `curr` hier ein
  - sonst: verschmelze die Bäume und füge den neuen Baum bei der Position `rang(curr) + 1` ein $\rightarrow$ evtl. Überlauf behandeln

Consolidate: initial

0	1	2	3
–	–	–	–

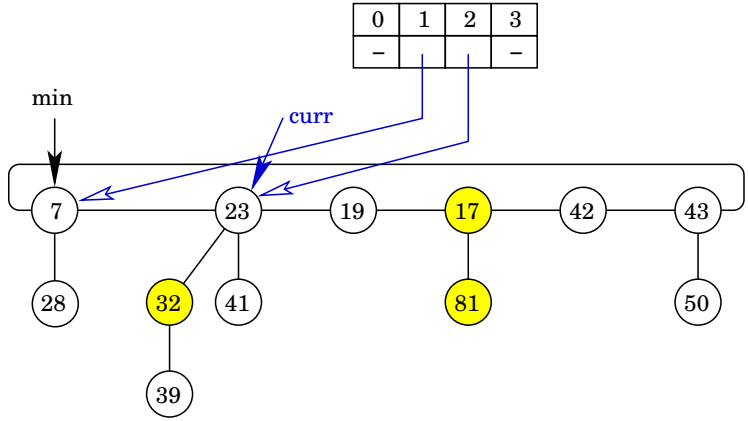


Consolidate: trage Baum mit Rang 1 in das Array ein



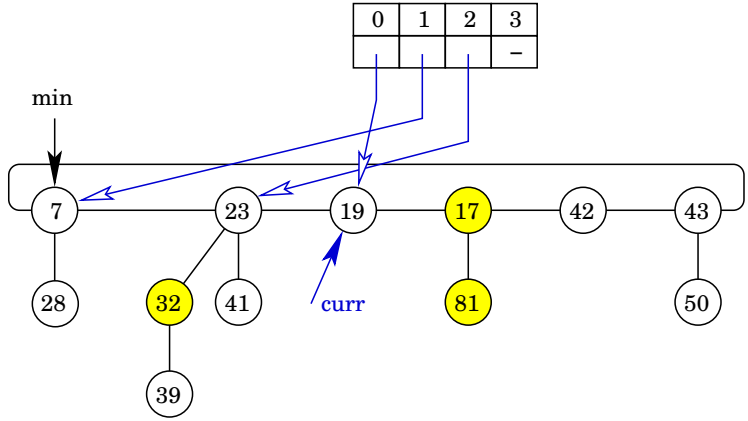
dann: setze Zeiger `curr` auf das nächste Element der Wurzelliste

Consolidate: trage Baum mit Rang 2 in das Array ein



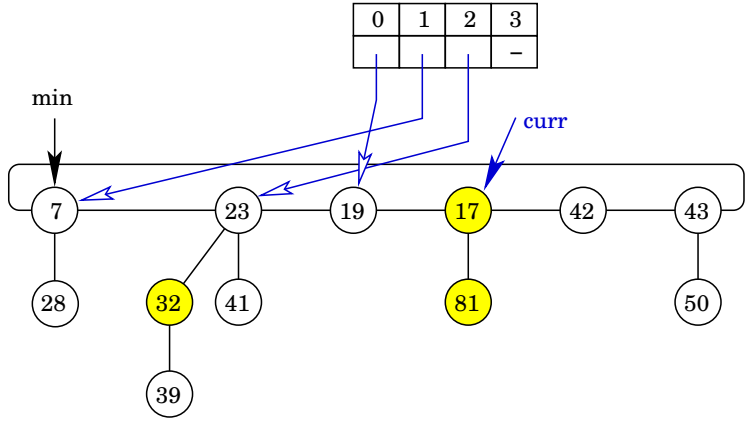
dann: setze Zeiger `curr` auf das nächste Element der Wurzelliste

Consolidate: trage Baum mit Rang 0 in das Array ein



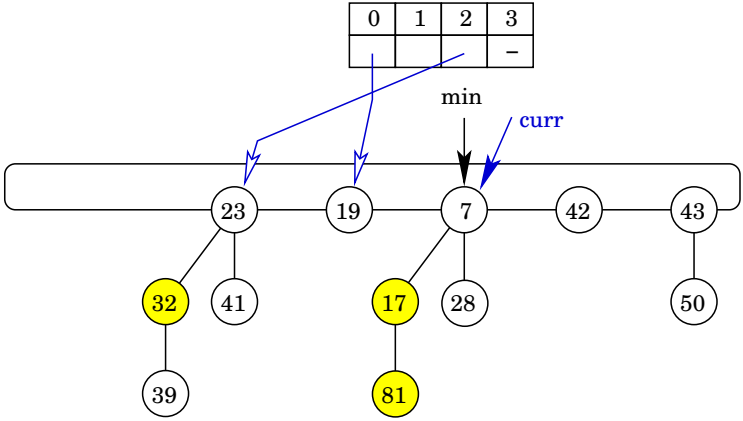
dann: setze Zeiger `curr` auf das nächste Element der Wurzelliste

Consolidate: `A[1]` ist bereits belegt



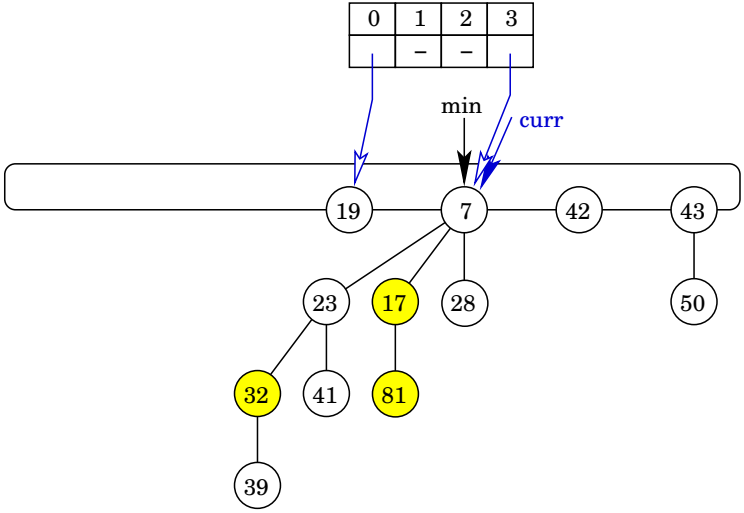
also: verschmelze die Bäume mit Wurzeln 7 und 17

Consolidate: `A[2]` ist bereits belegt → Überlauf behandeln



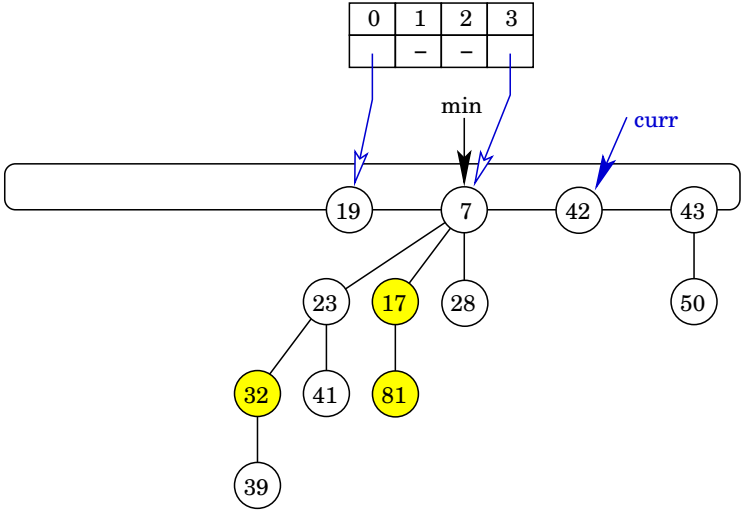
also: verschmelze die Bäume mit Wurzeln 7 und 23

Consolidate: trage neuen Baum mit Rang 3 in das Array ein



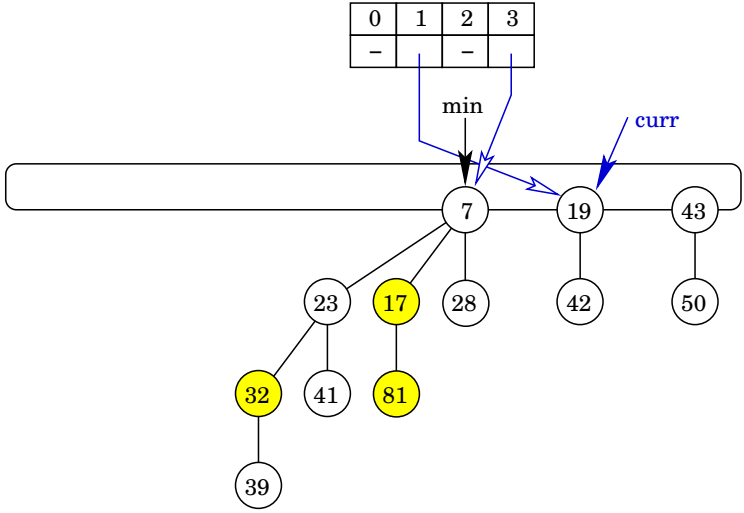
dann: setze Zeiger `curr` auf das nächste Element der Wurzelliste

Consolidate: `A[0]` ist bereits belegt



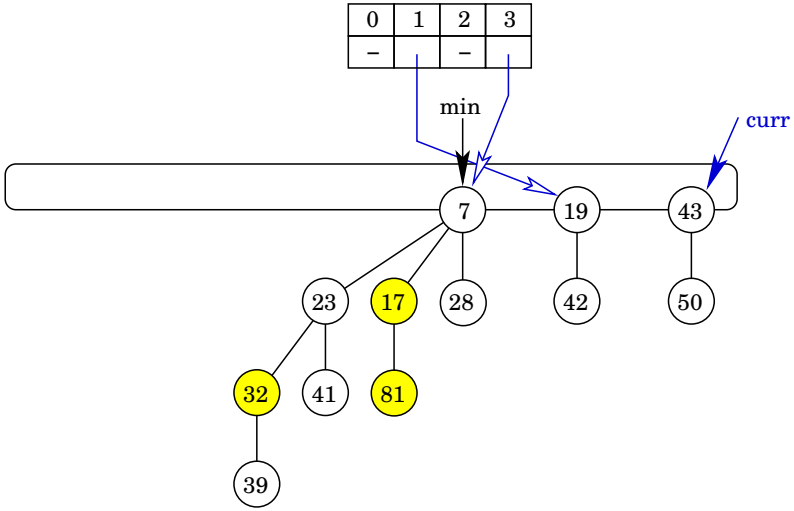
also: verschmelze die Bäume mit Wurzeln 19 und 42

Consolidate: trage neuen Baum mit Rang 1 in das Array ein



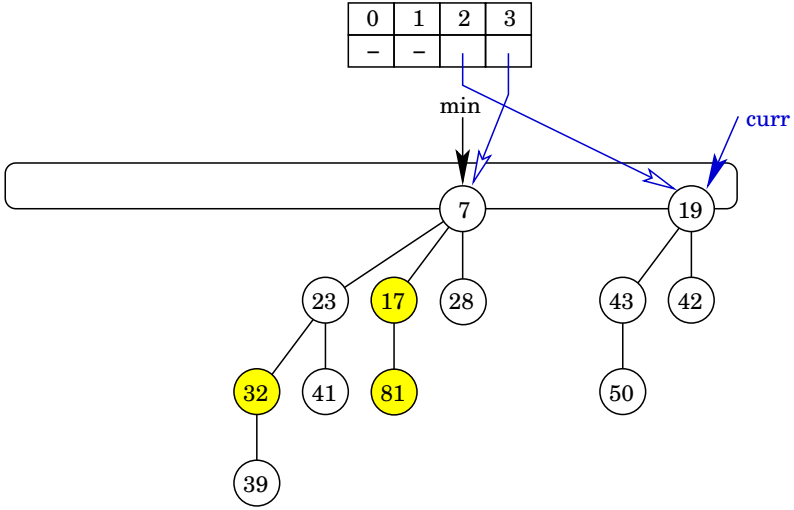
dann: setze Zeiger curr auf das nächste Element der Wurzelliste

Consolidate: `A[1]` ist bereits belegt



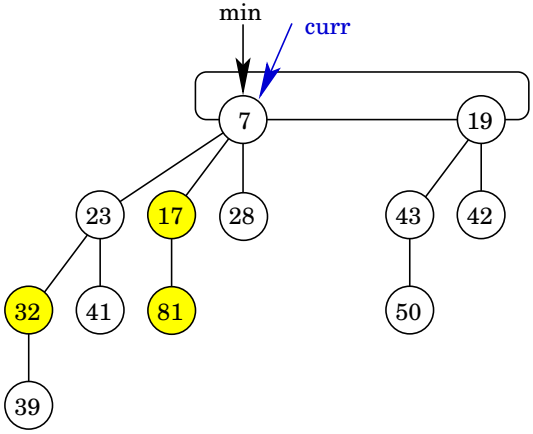
also: verschmelze die Bäume mit Wurzeln 19 und 43

Consolidate: trage neuen Baum mit Rang 2 in das Array ein



dann: setze Zeiger `curr` auf das nächste Element der Wurzelliste

Consolidate: Ende, denn `curr` zeigt wieder auf `min`



Laufzeit?

DecreaseKey(H, x, k):

- Trenne x von seinem Elter.
- Verkleinere den Wert auf k.
- Hänge den heap-geordneten Baum in die Wurzelliste.
- Aktualisiere den Minimalzeiger.

$\rightarrow$ Laufzeit: $\Theta(1)$

Wie verhindern wir, dass die Bäume zu „dünn“ werden, also zu viele Knoten verlieren?

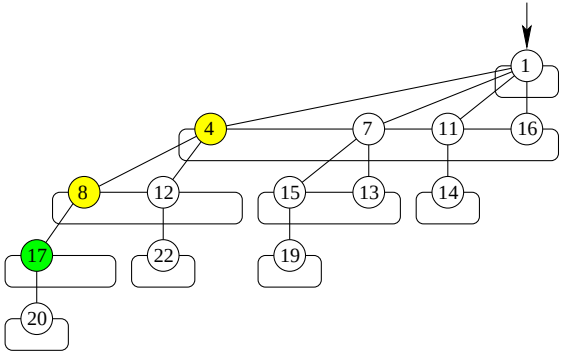
Es soll verhindert werden, dass mehr als zwei Kinder von seinem Elter abgetrennt werden:

- Beim Abtrennen eines Knotens p von seinem Elter v wird v markiert.
- War v bereits markiert, wird auch v von seinem Elter v' abgetrennt und v' markiert, usw.
- Alle abgetrennten Teilbäume werden in die Wurzelliste aufgenommen, die Markierungen der jeweiligen Wurzeln werden gelöscht.

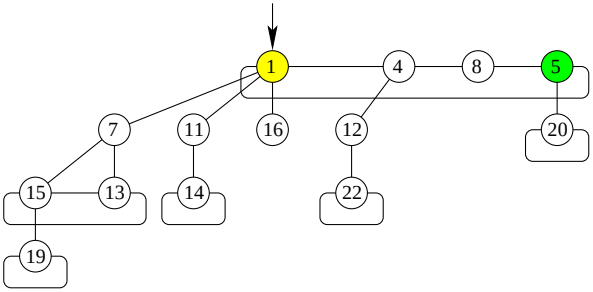
Laufzeit:

- $\mathcal{O}(\log(n))$ im worst-case
- $\mathcal{O}(1)$ amortisiert

Beispiel:



decreaseKey(17, 5)



Delete(H, x): Setze x auf einen sehr kleinen Schlüssel mittels DecreaseKey und führe dann ExtractMin(H) aus.

- $\mathcal{O}(n)$ im worst-case
- $\mathcal{O}(\log(n))$ amortisiert

Amortisierte Laufzeit:

- Durchschnittliche, maximale Kosten einer Operation bei einer beliebigen Folge von Operationen.
- Idee: Eine einzelne ExtractMin-Operation kann zwar sehr lange dauern, aber dann wird die Datenstruktur „aufgeräumt“ und die nächste Ausführung dauert dann nicht so lange.

Zusammenfassung

Operation	Linked List(*)	Binary Heap(*)	Binomial Heap(*)	Fibonacci Heap(**)
MakeHeap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
DecreaseKey	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

(*) worst-case-Laufzeit

(**) amortisierte Laufzeit

Übung 55.

- *Fügen Sie die Werte 1,2,3,4,5,6,7 in dieser Reihenfolge in einen initial leeren Fibonacci-Heap ein.*
- *Führen Sie anschließend* ExtractMin() *und danach* DecreaseKey(6, 1) *aus.*
- *Geben Sie nach jedem Schritt den resultierenden Fibonacci-Heap an.*

Übersicht

Motivation

Immer größere Datenmengen werden gesammelt und verarbeitet:

- geografische Informationssysteme
- Molekularbiologie (DNA-Sequenzen), Bioinformatik
- Suche im Web

Probleme:

- Geschwindigkeit der Prozessoren verbessert sich pro Jahr um etwa 30% - 50%, Geschwindigkeit des Speichers nur um 7% - 10% pro Jahr
- Betriebssysteme versuchen durch *caching* und *prefetching* I/O-Engpässe zu vermeiden (virtuelles Speichermanagement), aber: diese allgemeinen Algorithmen sind nicht speziell auf das jeweilige Problem optimiert
- Zugriff auf Hauptspeicher spricht kleine Blöcke an (cache-line 64 Byte bei Intel Core i7), Zugriff auf externen Speicher wie HDD/SSD spricht große Blöcke an

$\rightarrow$ greifen Algorithmen unstrukturiert auf Sekundärspeicher zu, ohne Lokalität der Zugriffe auszunutzen, erfolgen viel mehr Speicherzugriffe als nötig

Durchwandern eines Arrays

Initialisieren:

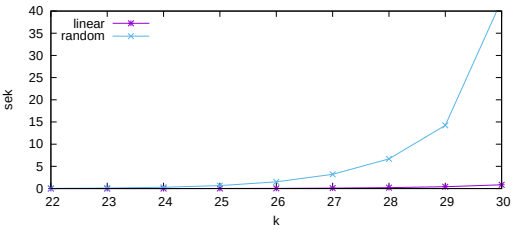
```
for (i = 0; i < N; i++)
     D[i] = i;
C = permute(D)
```

Lineares Durchlaufen:

```
for (i = 0; i < N; i++)
    A[D[i]] = A[D[i]] + 1;
```

Zufälliges Durchlaufen:

```
for (i = 0; i < N; i++)
    A[C[i]] = A[C[i]] + 1;
```



Laufzeiten für Arrays mit $N = 2^k$ vielen `int`-Elementen, $k = 22, \ldots, 30$ auf einem PC mit Intel Core i7, 2.80 GHz CPU, 16 GB RAM, Ubuntu 20.04 LTS.

Hierarchisches Speichermodell[27] für Intel Core i7:

- 1 cycle to read a CPU register
- 4 cycles to reach L1-Cache (32 kB + 32 kB, 64 byte per line)
- 11 cycles to reach L2-Cache (256 kB, 64 byte per line)
- 39 cycles to reach L3-Cache (2 MB per core (shared), 64 byte per line)
- ≈100 cycles to reach main memory (RAM)
- ≈10.000 of cycles to reach external memory

Wir müssen also versuchen unsere Programme so zu schreiben, dass die Daten möglichst im Cache vorhanden sind, wenn wir sie benötigen. Das funktioniert nicht immer, aber wir werden sehen, dass oft ein bisschen Nachdenken hilft, um die Programme signifikant zu beschleunigen.

Größen in einem 64-bit C-Programm: 64 byte per line = 16 `int` = 8 `long int`

[27] Ulrich Drepper: What every programmer should know about memory. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`

Speicherhierarchie ist sehr komplex und besitzt viele Abstufungen:

- Hardware Caches nutzen verschiedene, komplexe Ersetzungsstrategien
- Swap-Bereich: Mapping der virtuellen auf physikalische Speicheradressen erfolgt mittels Translation Lookaside Buffer.
- Vielzahl von Parametern resultiert in komplexen Modellen
- komplexe Modelle erschweren Analyse von Algorithmen und Datenstrukturen

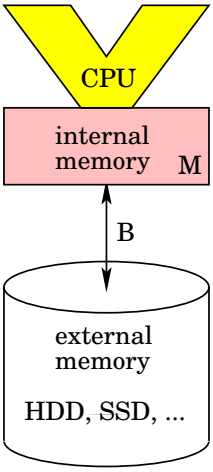
$\rightarrow$ externes Speichermodell als Kompromiss

externes Speichermodell

- starke Abstraktion der Speicherhierarchie auf nur 2 Ebenen
- wird oft verwendet für theoretische Analyse und Entwicklung von Algorithmen und Datenstrukturen
- Vitter[28]: Ergebnisse des Modells sind in die Praxis übertragbar

[28] Vitter, Jeffrey Scott: External memory algorithms and data structures: Dealing with Massive Data. ACM Computing Surveys, 33(2):209–271, 2001.

Externes Speichermodell



Parameter des Modells:

- M maximale Anzahl Datenelemente in dem internen Speicher
- B Anzahl Datenelemente, die mit einer I/O-Operation zwischen dem internen und dem externen Speicher ausgetauscht werden kann

Annahmen für die Performanceanalyse:

- Bandbreite ist unendlich und damit nicht der Flaschenhals der Auslagerung
- Datenzugriff auf internen Speicher kostet eine Zeiteinheit
- arithmetische Operationen kosten eine Zeiteinheit

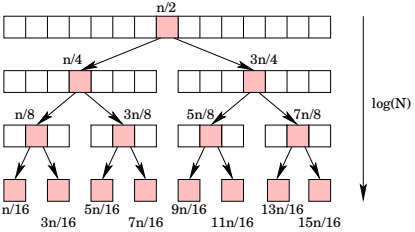
$\rightarrow$ Anzahl der I/O-Operationen ist aussagekräftig

Externer Speicher kann auch RAM bezeichnen, dann ist der interne Speicher der Cache. Das Modell kann also auch genutzt werden, um Cache-Effekte zu untersuchen.

Übersicht

Annahme: $n = 2^k - 1$ für ein $k \in \mathbb{N}$.

- zunächst müssen die Werte sortiert werden
- sortieren lohnt sich nur, wenn oft Werte gesucht werden
- bei jeder Suche wird zunächst auf das mittlere Element zugegriffen → Rang eins
- danach wird auf eines der mit Rang 2 markierten Elemente zugegriffen



Bei einem großen Array liegen das Rang-1-Element und die Rang-2-Elemente weit auseinander, also in verschiedenen Blöcken, sodass bei jedem Zugriff ein Cache-Miss auftritt.

Idee: platziere die Elemente, auf die oft zugegriffen wird, nah beieinander an den Anfang des Arrays

Rang:	4	3	4	2	4	3	4	1	4	3	4	2	4	3	4
Werte:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Rang:	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
Werte:	8	4	12	2	6	10	14	1	3	5	7	9	11	13	15
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Dazu muss aber eine völlig andere Sortierung der Daten vorgenommen werden und bei der binären Suche muss die Index-Berechnung des als nächstes zu untersuchenden Elements geändert werden.

Sortierung anpassen:

```
int i = 0;
for (int t = 2; t-1 <= N; t *= 2) {
    for (int p = 1; p < t; p += 2)
        r[i++] = s[p*N/t];
}
```

Binäre Suche

```
void *msearch(const void *key, const void *base, size_t nmemb,
        size_t size, int (*comp)(const void *, const void *)) {
    int p = 0;
    char *pos = (char *)base;

    while (p < nmemb) {
        int vgl = comp(key, pos);
        if (vgl == 0)
            return pos;
        if (vgl < 0) {
            pos += (p + 1) * size;
            p = 2*p + 1;
        } else {
            pos += (p + 2) * size;
            p = 2*p + 2;
        }
    }
    return nullptr;
}
```

Experiment:

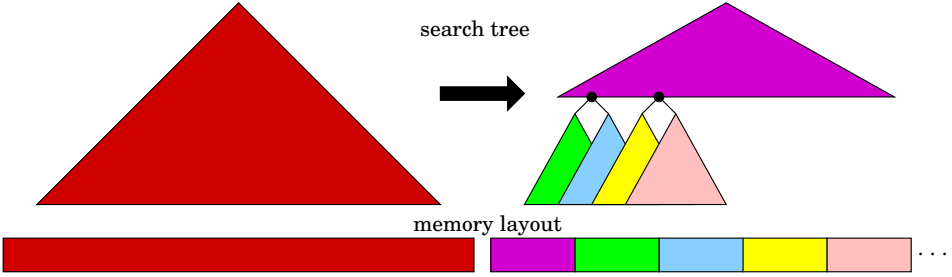
- erzeuge Array mit n Elementen vom Typ `int`
- `for (int i = 0; i < n; i++) A[i] = rand() % n`
- jedes Element von 0 bis n wird einmal gesucht und die Zeit in Sekunden gemessen
- vergleiche `msearch` mit `bsearch` aus der Standardbibliothek

n	qsort	bsearch	msearch
2^25 = 33554431	4	1	1
2^26 = 67108863	9	3	2
2^27 = 134217727	20	6	5
2^28 = 268435455	41	16	11
2^29 = 536870911	84	37	22
2^30 = 1073741823	198	86	46
2^31 = 2147483647	408	231	109

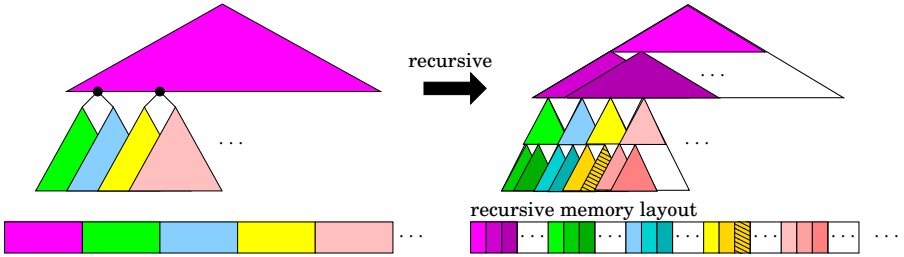
Problem: Die Lokalität der Daten wird nur für die ersten Zugriffe verbessert, weil nur die ersten Ebenen des „Suchbaums“ innerhalb eines Blocks liegen.

Besser: Layout nach van Emde Boas

- zunächst: erstelle vollständig balancierten Suchbaum
- teile den Baum in der Mitte bzgl. der Höhe



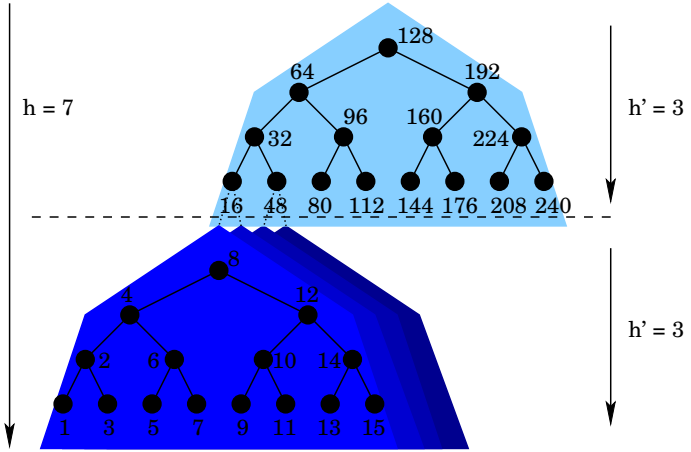
- platziere Teilbäume rekursiv im Speicher



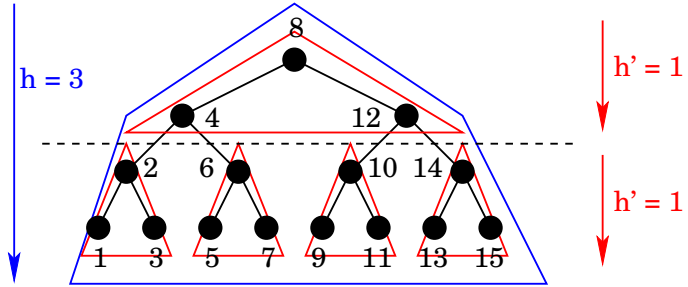
Ein Suchbaum der Höhe x hat 2^x viele Blätter und $2^{x+1} - 1$ viele Knoten.

- Suchbaum T habe Höhe $h = 2k + 1$, also $N = 2^{2k+2} - 1$ Knoten.
- Teilen wir T in der Mitte bzgl. der Höhe auf, erhalten wir oben einen Teilbaum der Höhe k und unten 2^{k+1} viele Teilbäume der Höhe k. Jeder dieser Teilbäume hat $2^{k+1} - 1$ Knoten.
- Insgesamt erhalten wir also $2^{k+1} - 1 + 2^{k+1} \cdot (2^{k+1} - 1) = 2^{2k+2} - 1$ viele Knoten.
- Anmerkung: Für $N = 2^x$ gilt $\sqrt{N} = 2^{x/2}$, nach der ersten Rekursion sind in den obigen Teilbäumen also etwa $\sqrt{N}$ viele Knoten enthalten.

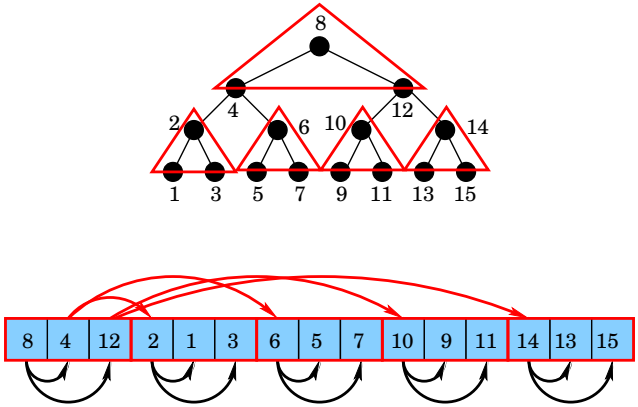
Beispiel: Für $N = 255$ erhalten wir einen Baum der Höhe $h = 7$. Teilen wir den Baum in der Mitte bzgl. der Höhe, so erhalten wir oben einen Baum der Höhe 3 mit 15 Knoten und unten 16 Bäume mit jeweils 15 Knoten, insgesamt also $17 \cdot 15 = 255$ Knoten.



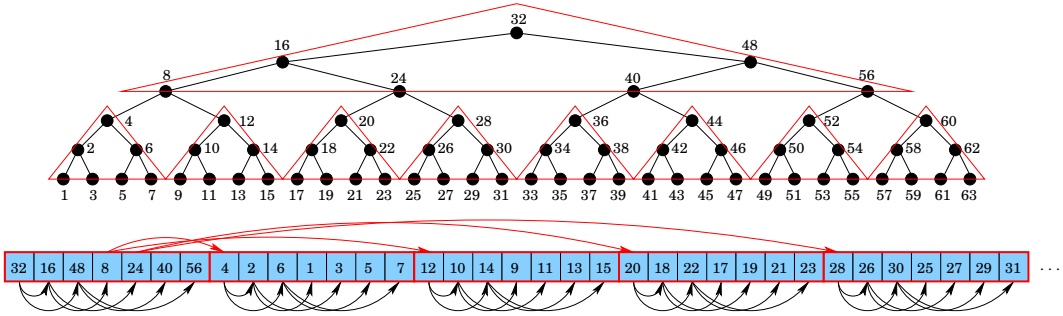
Für eine Blöckgröße von 3 Elementen müssen die Bäume der Höhe $h = 3$ aufgeteilt werden und es entstehen Bäume der Höhe 1, die jeweils 3 Knoten haben, insgesamt also $5 \cdot 3 = 15$ Knoten.



Für die Blockgröße 3 erhalten wir dann bspw. das folgende Layout:



Beispiel: Für eine Blockgröße 7 und $N = 63$ Elemente müssen wir den Baum nur einmal in der Mitte bzgl. der Höhe aufteilen und erhalten das folgende Layout:



Analyse: Betrachten wir einen Suchpfad von der Wurzel zu einem Blatt.

- die Länge des Pfades ist $\log_2(N)$
- bei einer Blockgröße B hat ein Teilbaum mit B Knoten die Höhe $\log_2(B)$
- beim Durchlaufen des Suchpfads müssen $\log_2(N)/\log_2(B)$ Blöcke gelesen werden
- es müssen $\log_B(N) = \log_2(N)/\log_2(B)$ viele I/Os durchgeführt werden

$\rightarrow$ Reduktion von $\log_2(N)$ auf $\log_B(N)$

Anwendung beim Rechnen mit Matrizen, z.B. bei der schnellen Fourier-Transformation.

Die i-te Zeile der transponierten Matrix A^T entspricht der i-ten Spalte der Matrix A, die transponierte Matrix A^T entsteht also durch Spiegelung der Matrix A an ihrer Hauptdiagonalen.

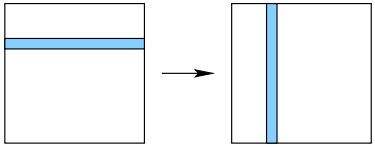
$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \end{pmatrix} \quad \rightarrow \quad A^T = \begin{pmatrix} 1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 \\ 2 & 6 & 10 & 14 & 18 & 22 & 26 & 30 \\ 3 & 7 & 11 & 15 & 19 & 23 & 27 & 31 \\ 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 \end{pmatrix}$$

Wir wollen uns hier ansehen, wie man durch Ausnutzen von Cache-Effekten das Transponieren einer Matrix effizient implementieren kann.

Vereinfachung: Wir betrachten nur quadratische Matrizen, also $n = m$.

Im einfachsten Fall können wir das Transponieren einer Matrix A mittels zweier verschachtelter Schleifen realisieren:

```
for (int r = 0; r < n; r++)
    for (int c = 0; c < n; c++)
        AT[c][r] = A[r][c];
```



Annahme:

- Die Größe eines Blocks beträgt B Elemente, es werden also bei einer I/O-Operation B Elemente der Matrix gelesen oder geschrieben. (1)
- Nicht alle n Elemente einer Zeile passen in den Cache. (2)

Analyse:

- eine Zeile aus A lesen $\rightarrow {}^{n}/_{B}$ viele I/O-Operationen wg. (1) und (2)
- eine Spalte nach A^T schreiben $\rightarrow n$ viele I/O-Operationen wg. (2)

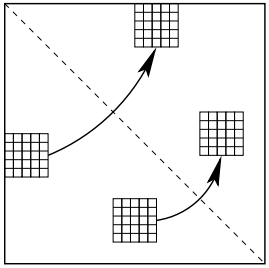
$\rightarrow$ insgesamt also $n \cdot ({}^{n}/_{B} + n) = n^2 + {}^{n^2}/_{B} \in \mathcal{O}(n^2)$ viele I/O-Operationen

Transponieren einer Matrix

Wir gehen daher anders vor: Zur Cache-Größe C definieren wir eine Blockgröße B, sodass $2B^2 \leq C$ gilt und daher beide Teilmatrizen in den Cache passen.

- Durchlaufe die Teilmatrizen der Größe $B \times B$ von A und transponiere diese Teilmatrizen.
- 1 Block von A lesen $\rightarrow B$ viele I/Os
- 1 Block nach A^T schreiben $\rightarrow B$ viele I/Os
- insgesamt gibt es $n/B \cdot n/B$ viele Teilmatrizen

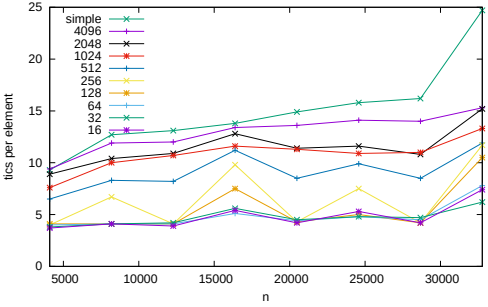
$\rightarrow$ $n^2/B^2 \cdot 2B = 2n^2/B \in \mathcal{O}(n^2/B)$ viele I/Os

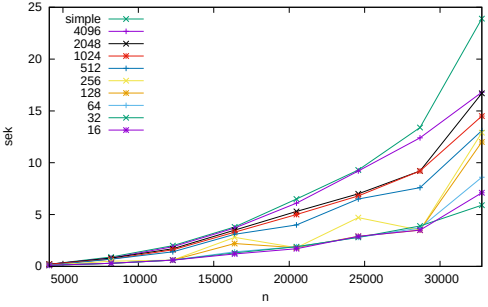


```
for (int i = 0; i < n; i += blocksize)
    for (int j = 0; j < n; j += blocksize)
        // transpose the block beginning at [i,j]
        for (int k = i; k < i + blocksize; ++k)
            for (int l = j; l < j + blocksize; ++l)
                AT[l][k] = A[k][l];
```

Welche Blockgrößen passen in welchen Cache?

- 32 kB großer L1-Cache: 64×64 `int` entspricht 16 kB (beste Resultate)
- 256 kB großer L2-Cache: 256×256 `int` entspricht 256 kB
- 8 MB großer L3-Cache: 1024×1024 `int` entspricht 4 MB





Wenn die Teilmatrizen nicht mehr in den Cache passen, dann ist die Performanz bei beiden Verfahren annähernd gleich schlecht.

Zur Vereinfachung beschränken wir uns wieder auf quadratische $n \times n$ Matrizen.

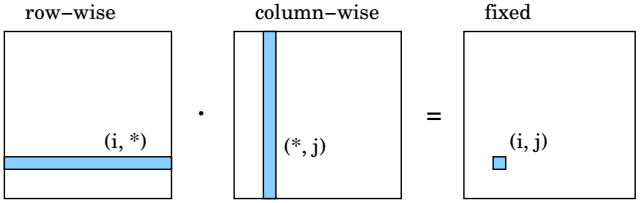
$$\begin{pmatrix} z_{11} & \cdots & z_{1n} \\ z_{21} & \cdots & z_{2n} \\ \vdots & \ddots & \vdots \\ z_{n1} & \cdots & z_{nn} \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \cdots & x_{2n} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nn} \end{pmatrix} \cdot \begin{pmatrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \cdots & y_{2n} \\ \vdots & \ddots & \vdots \\ y_{n1} & \cdots & y_{nn} \end{pmatrix} \text{ mit } z_{ij} = \sum_{k=1}^{n} x_{ik} \cdot y_{kj}$$

Da insgesamt n^2 viele Elemente z_{ij} der Ergebnismatrix zu berechnen sind und jede einzelne dieser Berechnungen einen Aufwand $\mathcal{O}(n)$ hat, ergibt sich insgesamt ein Aufwand von $\mathcal{O}(n^3)$ für $n \times n$ Matrizen bei Nutzung des naiven ijk-Verfahrens:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            Z[i][j] += X[i][k] * Y[k][j];
```

Problem der obigen, einfachen Matrix-Multiplikation:

- Wir nehmen wieder an, dass eine Zeile der Matrix nicht in den Cache passt.



- Für ein festes Paar (i, j), also bei der Berechnung eines Wertes z_{ij} gilt:
  - eine Zeile von X lesen $\rightarrow {}^{n}/_{B}$ viele I/O-Operationen
  - eine Spalte von Y lesen $\rightarrow n$ viele I/O-Operationen

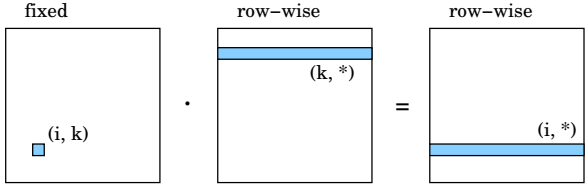
  $\rightarrow$ $n + {}^{n}/_{B}$ viele I/O-Operationen, um einen Wert z_{ij} zu berechnen
- Um n^2 Werte der Matrix Z zu berechnen werden also $n^2 \cdot (n + {}^{n}/_{B}) = n^3 + {}^{n^3}/_{B} \in \mathcal{O}(n^3)$ viele I/O-Operationen benötigt.

Durch einen einfachen Trick können wir die Anzahl der I/O-Operationen um etwa den Faktor $B/2$ reduzieren: Tausche die Zeilen 2 und 3 im obigen Code. → ikj-Verfahren

```
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            Z[i][j] += X[i][k] * Y[k][j];
```

Was erreichen wir dadurch?

- Für ein festes (i, k), also für den Durchlauf der inneren Schleife, muss jeweils auf eine Zeile aus Matrix Y und Z zugegriffen werden. → $2 \cdot n/B$ viele I/O-Operationen



- es gibt insgesamt n^2 viele Paare (i, k), also $n^2 \cdot 2n/B = 2n^3/B \in \mathcal{O}(n^3/B)$ viele I/Os

Der Wert der Matrix X, der in der innersten Schleife nicht geändert wird, kann in einer Variablen gespeichert werden, die der Compiler vermutlich in einem Register ablegt.

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        register int r = X[i][k];      // !!!!!

        for (int j = 0; j < n; j++)
            Z[i][j] += r * Y[k][j];
    }
}
```

Dadurch werden sehr teure Index-Zugriffe vermieden.

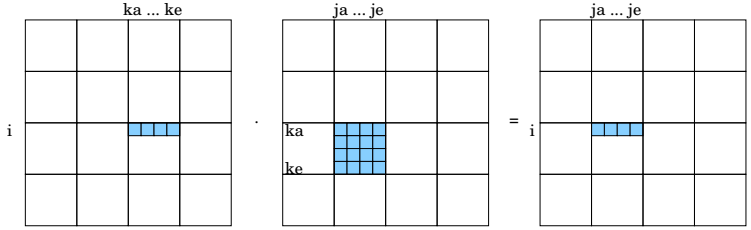
Es geht aber noch besser mit der gleichen Idee, die wir schon beim Transponieren einer Matrix kennen gelernt haben. Wir teilen die Matrizen in Teilmatrizen auf, sodass die Teilmatrizen in den Cache passen. Um die passende Blockgröße zu bestimmen, haben wir im folgenden Programm mit verschiedenen Blockgrößen experimentiert.

```
for (int ka = 0; ka < n; ka += blk) {
    int ke = min(ka + blk, n);
    for (int ja = 0; ja < n; ja += blk) {
        int je = min(ja + blk, n);
        for (int i = 0; i < n; i++) {
            for (int k = ka; k < ke; k++) {
                register int r = X[i][k];

                for (int j = ja; j < je; j++)
                    Z[i][j] += r * Y[k][j];
            }
        }
    }
}
```

Annahme: eine $B \times B$ Teilmatrix passt in den Cache

- Für ein festes Tupel (ka, ja, i) ergeben sich $B + 2$ viele I/O-Operationen:
  - um eine Blockzeile von X zu lesen $\rightarrow 1$ I/O
  - um einen ganzen Block von Y zu lesen $\rightarrow B$ I/O
  - um eine Blockzeile von Z zu schreiben $\rightarrow 1$ I/O



- Wird der Wert i geändert, ergibt sich ein Cache-Miss, weil eine Zeile nicht in den Cache passt. Für ein festes (ka, ja) ergeben sich also $n \cdot (B + 2)$ viele I/Os.
- Weil es $n/B \cdot n/B$ viele Paare (ka, ja) gibt, erhalten wir also insgesamt $n^2/B^2 \cdot n \cdot (B + 2) = n^3/B + 2n^3/B^2 \in \mathcal{O}(n^3/B)$ viele I/Os. (etwa Faktor 2 besser)

An den Laufzeiten sehen wir sehr schön das kubische Wachstum der Funktion:

$$T(n) \approx n^3 \quad \text{und} \quad T(2n) \approx (2n)^3 \;=\; 2^3 \cdot n^3 \;=\; 8n^3$$

n	ikjf	ikj	32	64	128	256	512	1024	ijk
512	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.2
1024	0.6	0.8	0.5	0.4	0.4	0.3	0.3	0.3	3.3
2048	5.8	6.8	4.4	3.7	3.4	2.8	2.4	2.6	65.3
4096	46.2	54.5	41.0	31.9	27.0	22.6	19.7	20.6	602.1
8192	371.9	441.0	381.0	262.0	212.9	232.8	211.4	188.4	6671.4

Bewertung:

- Das ijk-Verfahren sollte nach unserer Analyse etwa um den Faktor $B/2 = 16/2 = 8$ langsamer sein als das ikj-Verfahren.
- Das geblockte Verfahren sollte etwa um Faktor 2 schneller sein als ikj-Verfahren

Übersicht

Externe Datenstrukturen

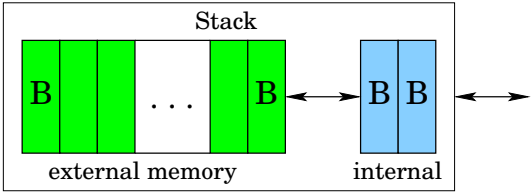
Große Datenmengen können nicht im Hauptspeicher verwaltet werden, daher müssen Teile der Daten auf einen sekundären oder externen Speicher wie HDD, SSD oder Flash-Medien ausgelagert werden.

Die Seitenersetzungsverfahren wie LRU der Betriebssysteme sowie das Caching und Prefetching sind allgemeine Verfahren, die nicht speziell auf die jeweiligen Probleme abgestimmt sind.

Hier: Spezielle (einfache) Datenstrukturen, um große Datenmengen mit wenigen I/O-Zugriffen verwalten zu können.

- Stack
- Queue
- Lineare Liste
- B-Baum
- externes Hashing
- Array-Heap

Nutze kombinierten Input/Output-Buffer der Größe $2 \cdot B$:



push
- füge neues Element in den Buffer ein
- falls Buffer voll ist: schreibe unteren Block auf die Festplatte

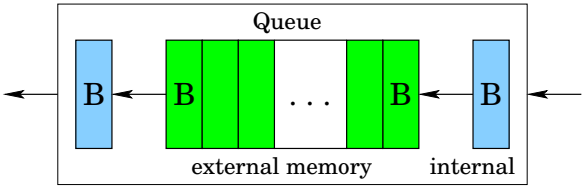
$\rightarrow$ $1/B$ viele I/Os amortisiert

top/pop
- lese/entferne Element aus dem Buffer
- falls Buffer leer ist: lese Block von der Festplatte in den Buffer ein

$\rightarrow$ $1/B$ viele I/Os amortisiert

Frage: Warum hat der Buffer eine Größe von $2 \cdot B$?

Nutze getrennte Input- und Output-Buffer jeweils der Größe B:



put
- füge neues Element am Ende des Input-Buffers ein
- falls Input-Buffer voll ist: schreibe Buffer auf die Festplatte

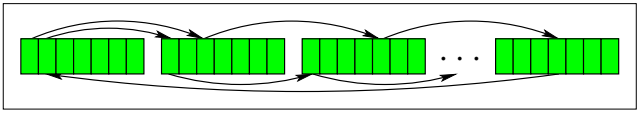
$\rightarrow$ $1/B$ viele I/Os amortisiert

head/get
- lese/entferne erstes Element aus dem Output-Buffer
- falls Output-Buffer leer ist:
  - lese Block von der Festplatte oder,
  - falls leer, aus dem Input-Buffer

$\rightarrow$ $1/B$ viele I/Os amortisiert

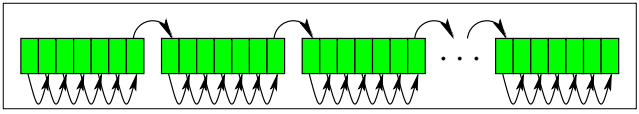
Versuch 1:

- einfach oder doppelt verkettete Elemente können so im Speicher verteilt liegen, dass aufeinanderfolgende Elemente in unterschiedlichen Blöcken liegen
- direkte Implementierung liefert pro Zugriff eine I/O-Operation



Versuch 2:

- speichere B konsekutive Listenelemente in einem Block und verkette die Blöcke

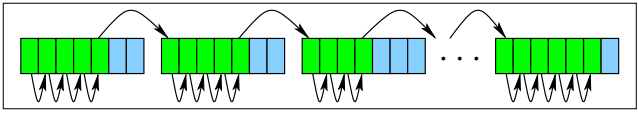


- ein Scan über alle n Elemente benötigt n/B viele I/O-Operationen ☺
- einfügen oder löschen eines Elements benötigt noch einmal n/B viele I/Os ☹

besser: Blöcke müssen nicht komplett gefüllt sein

- stelle sicher: *jedes Paar konsekutiver Blöcke* enthält mindestens $2/3B$ viele Elemente

→ maximal $3n/B \in \mathcal{O}(n/B)$ viele Blöcke werden zur Speicherung der n Werte benötigt



insert

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}(n/B)$ viele I/Os
- das eigentliche Einfügen kostet meist nur 1 I/O, außer wenn Block voll ist, dann:
  - falls benachbarter Block noch Platz hat: tausche ein Element aus $\rightarrow \mathcal{O}(1)$ viele I/Os
  - falls beide Nachbarblöcke voll sind:
    - teile den Block in 2 Teile der Größe $\approx B/2 \rightarrow \mathcal{O}(1)$ viele I/Os
    - danach sind mindestens $B/6$ Lösch-Operationen nötig, um Invariante zu verletzen

→ $\mathcal{O}(1 + n/B)$ viele I/Os

erase

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}(n/B)$ viele I/Os
- das eigentliche Löschen kostet meist nur 1 I/O, außer wenn der Block danach mit einem der beiden benachbarten Blöcke weniger als $2/3B$ Elemente besitzt
- dann: verschmelze die beiden Blöcke $\rightarrow \mathcal{O}(1)$ viele I/Os

$\rightarrow \mathcal{O}(1 + n/B)$ viele I/Os

Frage: Warum nicht einfach fordern, dass jeder Block mindestens zu einem Drittel gefüllt sein muss? Warum die „seltsame“ Regel mit Paaren von konsekutiven Blöcken?

B-Bäume und B*-Bäume sind externe Datenstrukturen. Sei b die Anzahl der Schlüssel, die in einem Knoten abgelegt werden können. Für $B = 4096$ Bytes lassen sich inklusive der zugehörigen Zeiger etwa

- $b = 256$ Elemente vom Typ `long int` oder
- $b = 340$ Elemente vom Typ `int` speichern.

unter Linux:

- `sudo tune2fs -l /dev/sda1` liefert Blockgröße des Dateisystems
- `sudo fdisk -l /dev/sda` liefert Größe der Sektoren

C++

- `sizeof(uintptr_t)` 8 Byte
- `sizeof(size_t)` 8 Byte
- `sizeof(int)` 4 Byte
- `sizeof(long int)` 8 Byte

Bei obigen Werten b und B können in einem B-Baum der Höhe h also etwa 256^{h+1} viele `long int`-Werte abgespeichert werden, wobei maximal h viele I/Os nötig sind.

Höhe	ungefähre Anzahl Elemente	
3	$16384 \cdot 2^{10}$	(16384 Kilo oder 16 Mega)
4	$4096 \cdot 2^{20}$	(4096 Mega oder 4 Giga)
5	$1024 \cdot 2^{30}$	(1024 Giga oder 1 Tera)
6	$256 \cdot 2^{40}$	(256 Tera)
7	$65536 \cdot 2^{40}$	(65536 Tera oder 64 Peta)
8	$16384 \cdot 2^{50}$	(16384 Peta oder 16 Exa)

Suchen, Einfügen und Löschen benötigen $\mathcal{O}(\log_b(N))$ viele I/Os.

Bereichsanfragen lassen sich mit B*-Bäumen aufgrund der verketteten Blätter sehr effizient realisieren.

Hashing = Schlüsselsuche durch Berechnung der Array-Indizes!

Idee:

- Notation: $[m] := \{0, 1, \ldots, m-1\}$
- bei einer Menge K von Werten aus einem Universum $U = [m]$, also $K \subseteq U$,
- verwende ein Array $A[0 \ldots m-1]$ und speichere die Schlüssel wie folgt:

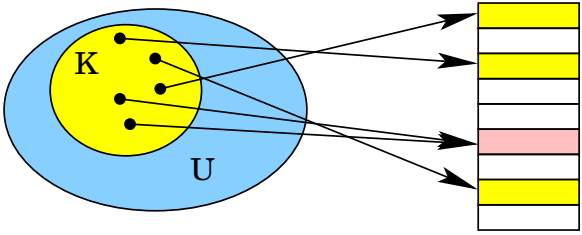
$$A[k] = \begin{cases} x & \text{falls } x.key \in K \text{ und } x.key = k \\ nil & \text{sonst} \end{cases}$$

$\rightarrow$ suchen, einfügen und löschen in $\mathcal{O}(1)$ Schritten

Probleme: Der Wertebereich $[m]$ kann sehr groß sein. Für 8-stellige Namen ergeben sich ungefähr $26^8 \approx 208 \cdot 10^9$ viele mögliche Werte. Außerdem wird viel Speicherplatz verschwendet, weil in der Regel nur wenige Plätze im Array genutzt werden.

[29] Wiederholung aus ALD

Lösung: Verwende Hash-Funktion h, um Wertebereich U und insbesondere die Menge der Schlüsselwerte $K \subseteq U$ auf Zahlen in $[n]$, $n \lll m$, abzubilden, also $h : U \to [n]$.



Anmerkungen:

- Die Hash-Funktion ist im Allg. nicht injektiv, d.h. verschiedene Schlüssel werden auf dieselbe Hashadresse abgebildet. $\to$ Adress-Kollision
  - Schlüssel, die auf die gleiche Adresse abgebildet werden, heißen *Synonyme*.
  - Die Menge der Synonyme bezüglich einer Adresse heißt *Kollisionsklasse*.
- Da die Hash-Funktion zum Platzieren und Suchen verwendet wird, muss sie einfach bzw. effizient zu berechnen sein.

Güte der Hash-Funktion h hängt von der gewählten Schlüsselmenge K ab.

- Güte ist nur unzureichend analysierbar
- zu h lässt sich immer ein K finden mit besonders vielen Kollisionen
- keine Hash-Funktion ist immer besser als alle anderen

Probleme treten auf

- beim Einfügen, wenn die berechnete Hashadresse nicht leer ist
- bei der Suche, wenn der berechnete Platz ein anderes Element enthält

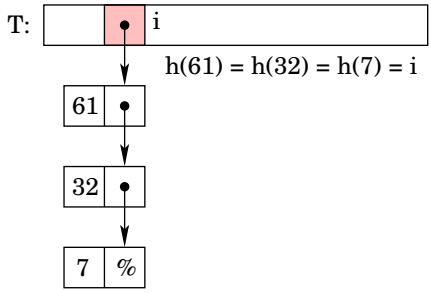
Kollisionsbehandlung für k_p falls $h(k_q) = h(k_p)$ für ein bereits gespeichertes k_q gilt:

- füge k_p in einem separaten Überlaufbereich außerhalb der Hashtabelle ein
  $\rightarrow$ *Verkettung der Überläufer*
- füge k_p an einem freien Platz innerhalb der Hashtabelle ein
  $\rightarrow$ *offenes Hashing* lineares oder quadratisches Sondieren, Double-Hashing

Hashing: Verkettung der Überläufer

Kollisionsauflösung:

- Überläufer werden in linearer Liste verkettet
- Liste wird an Hashtabelleneintrag angehängt
- Verkettung der Synonyme (Überläufer) pro Kollisionsklasse
- Suchen, Einfügen und Löschen sind auf eine Kollisionsklasse beschränkt



Durchschnittliche Anzahl der Einträge in $h(k)$ ist N/M, wenn N Einträge durch die Hash-Funktion gleichmäßig auf M Listen verteilt werden. *Belegungsfaktor:* $\alpha = N/M$

Kosten von Zugriffen sind abhängig vom Belegungsfaktor.

Problem: dynamische Speicheranforderung ist teuer

Idee: Speichere Überläufer in der Hashtabelle, nicht in zusätzlichen Listen.

- Ist Hashadresse $h(k)$ belegt, suche systematisch eine Ausweichposition.
- *Sondierungsfolge:* Folge der zu betrachtenden Speicherplätze für einen Schlüssel.
- Die Hash-Funktion hängt nun vom Schlüssel und von der Anzahl durchgeführter Platzierungsversuche ab:

$$h : U \times [m] \to [m]$$

- *wichtig:* Die Sondierungsfolge muss eine Permutation der Zahlen $0, \ldots, m-1$ sein, damit alle Einträge der Hashtabelle genutzt werden können.

Anmerkungen:

- Beim Einfügen und Suchen wird dieselbe Sondierungsfolge durchlaufen.
- Beim Löschen wird der Datensatz nicht gelöscht, sondern nur als gelöscht markiert: Der Wert wird ggf. bei einem späteren Einfügen überschrieben.
- Je voller die Tabelle wird, umso schwieriger wird das Einfügen neuer Schlüssel.

lineares Sondieren: Zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k, i) = (h'(k) + i) \bmod M.$$

quadratisches Sondieren: Zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k, i) = (h'(k) + (-1)^i \cdot \lceil i/2 \rceil^2) \bmod M.$$

Double-Hashing: Zu zwei gegebenen Hash-Funktionen $h_1(k)$ und $h_2(k)$ sei

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod M.$$

Knuth: We want to minimize the number of accesses to the file, and this has two major effects on the choice of algorithms:

- It is reasonable to spend more time computing the hash function, since the penalty for bad hashing is much greater than the cost of the extra time needed to do a careful job.
- The records are usually grouped into pages or buckets, so that several records are fetched from the external memory each time.

The file is divided into M buckets containing B records each. Collisions now cause no problem unless more than B keys have the same hash address.

$\rightarrow$ Hashing mit Verkettung der Überläufer oder offenes Hashing mit linearem Sondieren weisen im Erwartungsfall auch eine gute Performance im externen Speicher auf.

Implementierung einer Vorrangwarteschlange (priority queue)

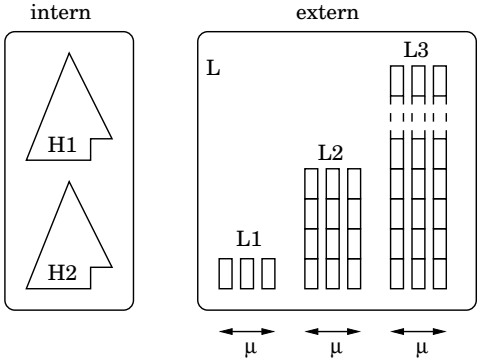
- intern: zwei Heaps plus zusätzlicher Speicher zum Mischen
- extern: Menge von sortierten Arrays unterschiedlicher Länge

Die Arrays sind in L Schichten L_i eingeteilt, $1 \leq i \leq L$.

Jede Schicht besteht aus $\mu = cM/B - 1$ vielen Slots der Länge $l_i = (cM)^i/B^{i-1}$ für $c < 1$.

Jeder Slot ist entweder leer oder enthält eine sortierte Folge von höchstens l_i Elementen.

Wegen $l_{i+1} = l_i \cdot (\mu + 1)$ gilt: Die Anzahl der Plätze in einem Slot von L_{i+1} entspricht der Anzahl aller Plätze in L_i plus l_i.



Invariante: Das kleinste Element befindet sich immer im internen Speicher.

Der interne Speicher ist aufgeteilt in die Heaps H_1 und H_2:

- H_1 hat die Größe $2cM$ und neue Elemente werden immer in H_1 eingefügt.
- H_2 enthält maximal die kleinsten B Elemente aus jedem belegten Slot j in Schicht L_i für alle $i = 1, \ldots, L$.

$\Rightarrow$ Es befinden sich maximal $2cM + B\mu L \leq 2cM + B \cdot {}^{cM}/_{B} \cdot L = cM \cdot (2 + L)$ viele Elemente im internen Speicher.

Zusätzlich wird Speicher für $(\mu + 1) \cdot B = cM$ benötigt, um die μ Slots plus eine Overflow-Folge zu mischen. Da M die Gesamtanzahl der Elemente im Hauptspeicher bezeichnet, muss $M \geq cM \cdot (3 + L)$ gelten:

$$M \geq cM \cdot (3 + L) \iff L \leq \frac{1 - 3c}{c}$$

Für $c = {}^{1}/_{7}$ gilt dann $L \leq 4$. $\rightarrow$ Der Wert L kann als Konstante betrachtet werden.

Operation insert: Fügt neues Element immer in den internen Heap H_1 ein.

Falls kein Platz in H_1 frei ist, dann verschiebe zunächst die $l_1 = cM$ größten Elemente in den externen Speicher.

- Falls ein freier Slot in L_1 existiert, dann verschiebe die $l_1 = cM$ größten Elemente in sortierter Reihenfolge in diesen freien Slot.

  Die kleinsten B dieser Elemente werden auch nach H_2 kopiert und verbleiben im internen Speicher.
- Sonst: Alle Elemente aus L_1 werden mit den $l_1 = cM$ größten Elementen aus H_1 zu einer sortierten Liste gemischt und dann in einen freien Slot von L_2 verschoben.
- Falls auch in L_2 kein freier Slot ist, dann wiederhole diesen Prozess solange, bis ein freier Slot gefunden wurde.

Operation extractMin:

Das kleinste Element befindet sich im internen Speicher, also in H_1 oder H_2. In beiden Fällen entfernt man das Element. War das kleinste Element in H_1, so ist nichts weiter zu tun.

Sonst korrespondiert das kleinste Element zu einem Element in Slot j einer Schicht L_i.

- Falls es das letzte in H_2 ist, das zu dem assoziierten Slot j der Schicht L_i gehört, dann werden die nächsten B Elemente von Slot j nach H_2 bewegt.
- Falls 2 Slots in L_i mit höchstens l_i Elementen existieren, inklusive der kleinsten Elemente aus H_2, dann werden die beiden Slots gemischt und in einen freien Slot von L_i eingetragen. Dadurch wird ein Slot in L_i frei. Außerdem werden die kleinsten B Elemente nach H_2 übertragen.

$\Rightarrow$ In jeder Schicht L_i existiert höchstens ein Slot, der weniger als $l_i/2$ Elemente enthält.

Externe Array-Heaps sind deutlich besser als B-Bäume zur Realisierung für Prioritätswarteschlangen geeignet[30].

Fragen:

- Welche Datenstrukturen eignen sich für die Heaps H_1 und H_2? (Es müssen keine echten Heaps sein.)
- Welche Datenstruktur eignet sich für die Slots und Schichten?
- Wie viele Elemente können maximal gespeichert werden, wenn $M = 10^8$, $B = 10^3$ und $c = 1/7$ ist?
- Wie sieht eine Folge von insert- und extractMin-Operationen aus, so dass der kleinste Wert des Array-Heaps in H_2 und nicht in H_1 enthalten ist?

[30] K. Brengel, A. Crauser, P. Ferragina, U. Meyer. An Experimental Study of Priority Queues in External Memory. ACM J. Exp. Algorithmics (5), 2000.

Übersicht

Übersicht

Motivation

```
class Stack {
protected:
   int _next, _size, *_values;

   bool isFull() {
      return _size == _next;
   }

   void enlarge() {
      int *tmp = new int[2 * _size];

      for (int i = 0; i < _size; i++)
          tmp[i] = _values[i];

      delete[] _values;
      _values = tmp;
      _size *= 2;
   }
```

```
public:
   Stack(int size = 8) {
      _size = size;
      _next = 0;
      _values = new int[size];
   }

   bool isEmpty() {
      return 0 == _next;
   }

   int top() {
      if (isEmpty())
         throw "empty stack exception";
      return _values[_next - 1];
   }
```

```
//    void pop() {
//       if (isEmpty())
//          throw "empty stack exception";
//       _next -= 1;
//    }

   void push(int val) {
      if (isFull())     // !!!!!!!!
         enlarge();     // !!!!!!!!

      _values[_next] = val;
      _next += 1;
   }
};
```

Laufzeit von `push` im worst-case:

- Speicherbereich für $2n$ Elemente allokieren.
- Verschiebe n Elemente in größeren Speicherbereich.
- Kopiere neues Element in den Stack.

$\rightarrow$ Laufzeit: $\mathcal{O}(n)$

N Ausführungen von `push` dauern also:

$$\sum_{i=1}^{N} \mathcal{O}(i) \;=\; \mathcal{O}(N^2)$$

Aber: Ein `push` ist nur selten teuer!

Genauere Abschätzung für N Ausführungen von `push`:

$$\sum_{i=1}^{N} 1 + \sum_{i=1}^{\log_2(N)} 2^i \;\leq\; N + 2^{\log_2(N)+1} \;=\; 3 \cdot N \;\in\; \mathcal{O}(N)$$

Amortisierte (durchschnittliche) Laufzeit von `push`:

$$T(N) \;=\; \frac{\mathcal{O}(N)}{N} \;=\; \mathcal{O}(1)$$

amortisieren: ausgleichen, aufwiegen

Aggregat-Methode

auch *Ganzheitsmethode* genannt:

- Zeige für jedes n,
- dass jede beliebige Folge von Operationen
- im worst-case
- eine Laufzeit von $T(n)$ hat.

$\rightarrow$ amortisierte Kosten pro Operation: $\frac{T(n)}{n}$

Problem: Keine Differenzierung auf unterschiedliche Typen von Operationen wie `push`, `pop`, `top`, usw.

Account-Methode

auch *Bankkonto-Paradigma* genannt:

- Betrachte eine beliebige Folge von n Operationen.
- Jede Operation kostet Geld.
- t_i: Tatsächliche Kosten der i-ten Operation.
- a_i: Amortisierte Kosten der i-ten Operation.
- falls $t_i < a_i$, dann schreibe Überschuss auf Konto gut.
- falls $t_i > a_i$, dann zahle Differenz vom Konto.
- Kontostand darf niemals negativ sein!

Pro `push` bezahle 3 elementare Einfüge-Operationen:

- einfügen des Elements
- verschieben des Elements beim Vergrößern
- verschieben eines anderen Elements beim Vergrößern

Der Kontostand wird niemals negativ! Nur bei der Vergrößerung des Speicherbereichs könnte der Kontostand negativ werden, aber:

- Zwischen den beiden Vergrößerungen auf 2^k und 2^{k+1} Elemente wurden genau 2^k Elemente eingefügt.
- Auf dem Konto ist also mindestens ein Guthaben von $2^k \cdot (3-1) = 2^{k+1}$.

$\rightarrow$ Entspricht genau der Anzahl Elemente, die verschoben werden müssen.

Potenzial-Methode

Anstelle von Kontoständen weise der Datenstruktur eine potenzielle Energie zu:

- Betrachte eine Folge von n Operationen.
- Ordne jedem Bearbeitungszustand ein nicht-negatives Potenzial zu.
- Ordne jeder Operation amortisierte Kosten zu.
- Φ_i: Potenzial nach Ausführung der i-ten Operation.
- Die amortisierten Kosten a_i der i-ten Operation sind die tatsächlichen Kosten t_i plus der Differenz der Potenziale.

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Die gesamten amortisierten Kosten sind:

$$\begin{aligned}
\sum_{i=1}^{n} a_i &= a_1 + a_2 + a_3 + \ldots + a_n \\
&= t_1 + \Phi_1 - \Phi_0 \\
&+ t_2 + \Phi_2 - \Phi_1 \\
&+ t_3 + \Phi_3 - \Phi_2 \\
&+ \quad \ldots \\
&+ t_n + \Phi_n - \Phi_{n-1} \quad = \sum_{i=1}^{n} t_i + \Phi_n - \Phi_0
\end{aligned}$$

Falls $\Phi_n \geq \Phi_0$ ist, dann gilt:

$$\sum_{i=1}^{n} t_i \;=\; \sum_{i=1}^{n} a_i - (\Phi_n - \Phi_0) \;\leq\; \sum_{i=1}^{n} a_i$$

Also: Für $\Phi_n \geq \Phi_0$ ist die Summe der amortisierten Kosten eine obere Schranke für den tatsächlichen Aufwand.

Problem: Wir wissen nicht, wie viele Operationen ausgeführt werden.

Daher: Stelle $\Phi_i \geq \Phi_0$ für alle i sicher.

- In der Praxis: $\Phi_0 = 0$ und $\Phi_i \geq 0$, d.h. die Potenzial-Funktion wird niemals negativ, oder anders gesagt:
- Wir zahlen wie beim Bankkonto-Paradigma im vorraus.

Für den Stack S definiere die Potenzial-Funktion Φ als:

$$\Phi(S) := 2 \cdot num(S) - size(S)$$

Dabei ist

- *num* die Anzahl der gespeicherten Elemente und
- *size* die Größe des Arrays.

Der Stack ist immer mindestens halb voll, also gilt:

$$num(S) \geq \frac{size(S)}{2} \iff 2 \cdot num(S) \geq size(S)$$

$\rightarrow \Phi(S)$ wird niemals negativ!

Wir unterscheiden zwei Fälle beim Einfügen des i-ten Elements:

- *Keine Vergrößerung des Arrays* $\rightarrow size_i = size_{i-1}$

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i - 1) - size_i) \\ &= 3 \end{aligned}$$

- *Array-Vergrößerung* $\rightarrow size_i/2 = size_{i-1} = num_i - 1$

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 3 \cdot num_i - size_i - (2 \cdot (num_i - 1) - size_i/2)) \\ &= num_i + 2 - size_i/2 \\ &= num_i + 2 - (num_i - 1) \;=\; 3 \end{aligned}$$

Bei pop wollen wir ungenutzten Speicher wieder freigeben.

```
void Stack::reduce() {
    .....
}

void Stack::pop() {
    if (isEmpty())
        throw "empty stack exception";
    _next -= 1;
    if (isSlightlyFilled())   // ?????
        reduce();             // !!!!!
}
```

Übung 56.

- *Ab welchem Füllgrad soll die* reduce*-Funktion aufgerufen werden?*
- *Welche amortisierte Laufzeiten haben dann die Funktionen* push *und* pop*?*

Lösung ???

erste Idee:

- Speicherplatz verdoppeln, wenn das Array voll belegt ist und ein weiteres Element eingefügt werden soll, und
- Speicherplatz halbieren, wenn Array weniger als halb voll ist.

funktioniert leider nicht:

- Füge $n = 2^k$ Werte ein: $num(S) = size(S) = n = 2^k$
- Betrachte die Folge $i, d, d, i,\ \ i, d, d, i,\ \ i, d, d, i, \ldots$ aus insert- und delete-Operationen, bzw. push- und pop-Operationen.
  - i: $num(S) = n + 1 \quad \rightarrow$ vergrößert auf $size(S) = 2^{k+1}$
  - d: $num(S) = n$
  - d: $num(S) = n - 1 \quad \rightarrow$ verkleinert auf $size(S) = 2^k$
  - i: $num(S) = n$

$\rightarrow$ Bei jeder zweiten Operation ergibt sich ein Aufwand von $\Theta(n)$, so dass die amortisierte Zeit pro Operation $\Theta(n)$ beträgt.

zweite Idee:

- Speicherplatz verdoppeln, wenn das Array voll belegt ist, und
- halbieren, wenn das Array nur noch zu einem Viertel belegt ist.

Problem: Die bisherige Potenzialfunktion

$$\Phi(S) = 2 \cdot num(S) - size(S)$$

kann dann negativ werden, z.B. wenn der Stack nur zu einem Drittel gefüllt ist, also wenn $num(S) = \frac{1}{3} \cdot size(S)$.

neue Potenzialfunktion:

$$\Phi(S) := \begin{cases} 2 \cdot num(S) - size(S), & \text{falls } \alpha(S) \geq \frac{1}{2} \\ size(S)/2 - num(S), & \text{sonst} \end{cases}$$

wobei $\alpha(S) := num(S)/size(S)$ der *Füllgrad*, engl. *load factor*, ist.

$$\Phi(S) := \begin{cases} 2 \cdot num(S) - size(S), & \text{falls } \alpha(S) \geq \frac{1}{2} \\ size(S)/2 - num(S), & \text{sonst} \end{cases}$$

Intuitiv: Φ gibt an, wie weit der Stack vom Füllgrad $\alpha = 1/2$ entfernt ist. Wenn $\alpha = 1/2$ ist, dann gilt $2 \cdot num(S) = size(S)$.

- Wenn der Stack halb gefüllt ist, also für $\alpha = 1/2$, ist der Wert der Potenzialfunktion 0.
- Wenn der Stack komplett gefüllt ist, also für $\alpha = 1$, dann gilt $size(S) = num(S)$ und der Wert der Potenzialfunktion ist $num(S)$. Das Potenzial reicht also, um das Kopieren aller Elemente zu bezahlen.
- Wenn der Stack nur noch zu einem Viertel gefüllt ist, also wenn $\alpha = 1/4$ ist, dann gilt $size(S) = 4 \cdot num(S)$, und der Wert der Potenzialfunktion beträgt $num(S)$. Somit kann das Kopieren aller Elemente aus dem Potenzial bezahlt werden.

Für die *pop-Operation* gilt $num_i = num_{i-1} - 1$, und wir müssen drei Fälle unterscheiden. Betrachten wir zunächst $\alpha_{i-1} < \frac{1}{2}$.

- keine Verkleinerung: Dann gilt $size_i = size_{i-1}$ und

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2 \end{aligned}$$

- Verkleinerung: Dann gilt $size_i/2 = size_{i-1}/4 = num_i + 1$ und

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= (\cancel{num_i} + 1) + (size_i/2 - \cancel{num_i}) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (num_i + 1) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1 \end{aligned}$$

Betrachten wir nun den Fall $\alpha_{i-1} \geq \frac{1}{2}$.

Dann wird das Array nicht verkleinert und es gilt $size_i = size_{i-1}$. Wir erhalten schließlich:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - \cancel{size_i}) - (2 \cdot (num_i + 1) - \cancel{size_i}) \\ &= 1 + 2 \cdot num_i - 2 \cdot (num_i + 1) \\ &= -1 \end{aligned}$$

Die amortisierten Kosten für eine pop-Operation sind also in allen Fällen durch eine Konstante nach oben beschränkt.

Für die *push-Operation* gilt $num_i = num_{i-1} + 1$.

Betrachten wir zunächst den Fall $\alpha_{i-1} \geq \frac{1}{2}$. Da für diesen Fall die alte und die neue Potenzialfunktion identisch sind, erhalten wir als Kosten für die push-Operation höchstens 3.

Für den Fall $\alpha_{i-1} < \frac{1}{2}$ kann keine Vergrößerung des Arrays notwendig werden, da eine Vergrößerung nur bei $\alpha_{i-1} = 1$ erfolgt. Also gilt $size_i = size_{i-1}$. Wir unterscheiden zwei Fälle.

- für $\alpha_i < \frac{1}{2}$ erhalten wir:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0 \end{aligned}$$

- für $\alpha_i \geq \frac{1}{2}$ erhalten wir:

$$\begin{aligned}
a_i &= t_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2 \cdot num_{i-1} + 2 - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 + 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&= 3 + 3 \cdot \alpha_{i-1} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&< 3 + \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&= 3
\end{aligned}$$

Auch für die push-Operation sind die amortisierten Kosten in allen Fällen durch eine Konstante nach oben beschränkt.

Mittels Bankkonto-Methode ergeben sich folgende amortisierten Laufzeiten:

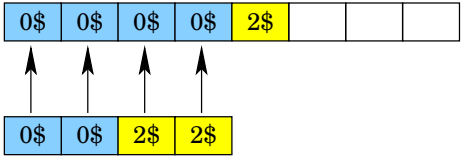
- push: $a_i = 3 \rightarrow 2\$$ sparen
- pop: $a_i = 2 \rightarrow 1\$$ sparen

Dann gilt für $size(S) = 2^{k+1}$ beim

- Vergrößern: Zwischen $num(S') = 2^k$ und $num(S) = 2^{k+1}$ gibt es 2^k push-Operationen mehr als pop-Operationen. Daher wurde ein Guthaben von mindestens $2 \cdot 2^k = 2^{k+1}$ angespart, wovon das Kopieren der 2^{k+1} Elemente in das nächst größere Array bezahlt werden kann.
- Verkleinern: Zwischen $num(S') = 2^{k-1}$ und $num(S) = 2^{k-2}$ gibt es 2^{k-2} pop-Operationen mehr als push-Operationen. Daher wurde ein Guthaben von mindestens 2^{k-2} angespart, wovon das Kopieren der 2^{k-2} Elemente in das kleinere Array bezahlt werden kann.

Account-Methode: push

Account-Methode: push

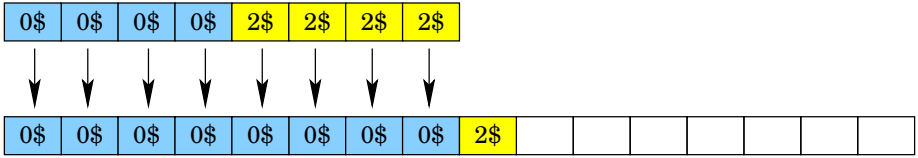


Account-Methode: push

Account-Methode: push

Account-Methode: push

Account-Methode: push



Account-Methode: pop

Account-Methode: pop

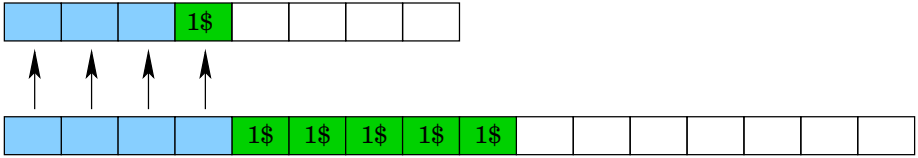
Account-Methode: pop

Account-Methode: pop

Account-Methode: pop

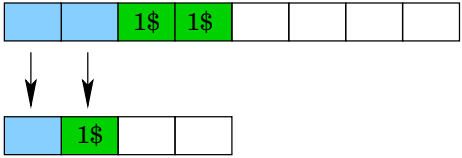
Account-Methode: pop

Account-Methode: pop



Account-Methode: pop

Account-Methode: pop



Übersicht

Motivation: Bei Datenbanken erfolgen 80% der Zugriffe auf nur 20% der Daten.

$\rightarrow$ Wie kann man den Zugriff effizient gestalten?

Aufgabenstellung:

- Gegeben: Linear verkettete Liste (unsortiert)
- Gesucht: Effizientes Einfügen, Löschen und Suchen.

Idee: Plaziere die Elemente, auf die häufig zugegriffen wird, weit vorne in der Liste.

Problem: In der Regel sind die Zugriffshäufigkeiten nicht im voraus bekannt.

Lösung: Ändere nach jedem Zugriff auf ein Element die Liste so, dass zukünftige Anfragen nach diesem Element schneller sind.

Die letzte Suchanfrage habe auf das Element e zugegriffen.

- *Move-To-Front-Regel* (MF-Regel)
  Mache e zum ersten Element der Folge.
- *Transpose-Regel* (T-Regel)
  Vertausche e mit unmittelbar vorangehendem Element.

Die relative Anordnung der anderen Elemente bleibt bei den obigen Regeln unverändert.

- *Frequency-Count-Regel* (FC-Regel)
  - Ordne jedem Element einen Häufigkeitszähler zu und
  - ordne die Liste nach jedem Zugriff entsprechend neu.

Beispiel: Betrachte Zugriffe auf die Zahlenfolge mittels *MF*-Regel:

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

- Greife zehnmal hintereinander auf alle Zahlen $1 \ldots 7$ zu:

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

  $\rightarrow$ Kosten pro Zugriff: 6.7
- Greife jeweils zehnmal auf jedes einzelne Element zu:

$$1, 1, 1, 1, 1, 1, 1, 1, 1, 1,\ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,\ \ldots$$

  $\rightarrow$ Kosten pro Zugriff: 1.3
- Ohne Selbstanordnung ergeben sich bei obigen Zugriffsfolgen jeweils die Kosten 4.

Gibt es eine optimale Strategie?

- Das Vorziehen eines Elements an den Listenanfang entsprechend der *MF*-Regel ist eine sehr drastische Änderung, die erst allmählich korrigiert werden kann, falls ein „seltenes“ Element „irrtümlich“ an den Listenanfang gesetzt wurde.
- Die *T*-Regel ist vorsichtiger und macht geringere Fehler, aber man kann leicht Zugriffsfolgen finden, so dass Zugriffe nach der *T*-Regel praktisch überhaupt nichts bringen.
- Die *FC*-Regel hat den Nachteil, dass man zusätzlichen Speicherplatz zur Aufnahme der Häufigkeitszähler bereitstellen muss.
- In der Literatur findet man weitere Permutationsregeln: *J.H. Hester und D.S. Hirschberg. Self-organizing linear search. ACM Computing Surveys, 17:295-311, 1985.*

- Wir müssen nicht nur unterschiedliche Zugriffshäufigkeiten berücksichtigen, sondern auch Clusterungen von Zugriffsfolgen, die sogenannte Lokalität.
- In der Literatur findet man meist nur asymptotische Aussagen über das erwartete Verhalten der Strategien, wobei die Zugriffsfolgen bestimmten Wahrscheinlichkeitsverteilungen genügen.
- Es gibt auch eine Reihe experimentell ermittelter Messergebnisse für reale Daten, bei denen man die verschiedenen Strategien relativ zueinander beurteilt.
- Sleator und Tarjan gelang ein bemerkenswertes theoretisches Ergebnis[31], das das sehr gute, beobachtete Verhalten der *MF*-Regel untermauert.

[31] D.D. Sleator and R.E. Tarjan: Amortized efficiency of list update and paging rules. Communications of the ACM, 28:202-208, 1985.

Selbstanordnende lineare Listen

Gegeben: Eine Liste mit N Elementen sowie eine Folge $s = s_1, \ldots, s_m$ von m Zugriffsoperationen.

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.

Kosten:

- Zugriff auf Element an Position k kostet k Einheiten.
- Vertauschung in Richtung Listenanfang ist kostenfrei.
- Vertauschung in andere Richtung kostet eine Einheit.

Zu einem Algorithmus A und einer Zugriffsoperation s_i definieren wir:

- $C_A(s_i)$ Schritte zur Ausführung der Zugriffsoperation s_i.
- $F_A(s_i)$ Anzahl kostenfreier Vertauschungen.
- $X_A(s_i)$ Anzahl kostenbehafteter Vertauschungen.

$C_A(s) = C_A(s_1) + \ldots + C_A(s_m)$, $F_A(s)$ und $X_A(s)$ analog.

Unsere bisher betrachteten Algorithmen

- *MF* (Move-to-Front),
- *T* (Transpose) und
- *FC* (Frequently-Count)

führen keine kostenbehafteten Vertauschungen durch. Somit gilt:

$$X_{MF}(s) = X_T(s) = X_{FC}(s) = 0$$

Wird auf ein Element an Position k zugegriffen, kann man es anschließend maximal mit allen $(k-1)$ vorangehenden Elementen kostenfrei vertauschen.

$\rightarrow$ Die Anzahl kostenfreier Vertauschungen ist höchstens so groß wie die Kosten der Operation minus 1.

Daher gilt für jede Strategie A (mit m Operationen):

$$F_A(s) \leq C_A(s) - m$$

denn

$$\begin{aligned} F_A(s) &= F_A(s_1) + F_A(s_2) + \ldots + F_A(s_m) \\ &\leq C_A(s_1) - 1 + C_A(s_2) - 1 + \ldots + C_A(s_m) - 1 \\ &= C_A(s) - m \end{aligned}$$

Definition: Seien L_1, L_2 Listen, die dieselben Elemente enthalten.

> $inv(L_1, L_2)$: Anzahl der Elementpaare x_i, x_j, deren Anordnung in L_2 eine andere ist als in L_1 → *Inversionen*

Beispiel: Für $L_1 = 4, 3, 5, 1, 7, 2, 6$ und $L_2 = 3, 6, 2, 5, 1, 4, 7$ erhalten wir $inv(L_1, L_2) = 12$, denn 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, 1 vor 4.

Diese Elementpaare stehen jedoch in L_1 in umgekehrter Reihenfolge.

Satz: Für jeden Algorithmus A der obigen Art und für jede Folge s von m Zugriffsoperationen gilt:

$$C_{MF}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Dieser Satz besagt grob, dass die MF-Regel höchstens doppelt so schlecht ist, wie jede andere Regel zur Selbstanordnung von Listen.

Die MF-Regel ist also nicht wesentlich schlechter als die beste überhaupt denkbare Strategie.

Beweis:

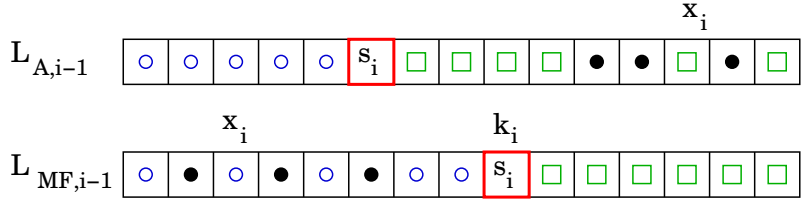
- Sei $L_{A,i}$ die von Algorithmus A geänderte Liste nach Zugriff i.
- Sei $L_{MF,i}$ die von der MF-Regel geänderte Liste nach Zugriff i.

Für die *Potenzialfunktion* $\Phi_i = inv(L_{A,i}, L_{MF,i})$ gilt:

- Initial ist $\Phi_0 = inv(L_{A,0}, L_{MF,0}) = 0$, da beide Algorithmen mit derselben Liste beginnen.
- Der Wert der Potenzialfunktion ist nie negativ!

Betrachte Zugriff auf das Element s_i:

- k_i: Position, an der Element s_i in $L_{MF,i-1}$ steht.
- x_i: Anzahl Elemente, die in $L_{MF,i-1}$ vor s_i aber in $L_{A,i-1}$ hinter s_i stehen.



Jedes dieser x_i Elemente ist mit s_i eine Inversion.

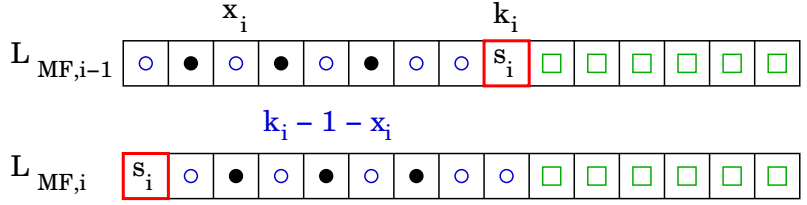
Es gilt:

$$inv(L_{A,i-1}, L_{MF,i}) \;=\; inv(L_{A,i-1}, L_{MF,i-1}) - x_i + (k_i - 1 - x_i)$$

denn: Durch das Vorziehen von s_i in $L_{MF,i-1}$

- verschwinden insgesamt x_i Inversionen und
- es entstehen $k_i - 1 - x_i$ Inversionen.

  $k_i - 1 - x_i$ ist die Anzahl der übrigen Elemente in $L_{MF,i-1}$ vor dem Element s_i.

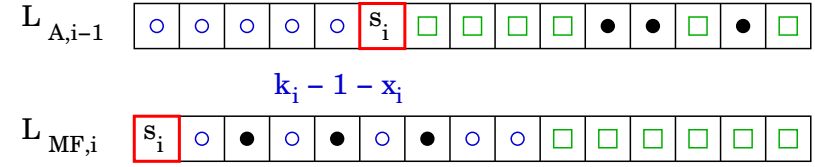


Es gilt also:

$$inv(L_{A,i-1}, L_{MF,i}) \;=\; inv(L_{A,i-1}, L_{MF,i-1}) - x_i + (k_i - 1 - x_i)$$

Anschließend:

- Jedes Vorziehen von s_i in $L_{A,i-1}$ mit Algorithmus A erniedrigt die Anzahl der Inversionen.
- Jedes Vertauschen von s_i in $L_{A,i-1}$ zum Listenende mit Algorithmus A erhöht die Anzahl der Inversionen.



Somit erhalten wir:

$$inv(L_{A,i}, L_{MF,i}) \;=\; inv(L_{A,i-1}, L_{MF,i-1}) - x_i + (k_i - 1 - x_i) - F_A(s_i) + X_A(s_i)$$

- Tatsächliche Kosten für Zugriff auf s_i mit MF-Regel: k_i

$\rightarrow$ Amortisierte Kosten a_i der i-ten Operation:

$$\begin{aligned} a_i &= k_i + inv(L_{A,i}, L_{MF,i}) - inv(L_{A,i-1}, L_{MF,i-1}) \\ &= k_i - x_i + (k_i - 1 - x_i) - F_A(s_i) + X_A(s_i) \\ &= 2(k_i - x_i) - 1 - F_A(s_i) + X_A(s_i) \end{aligned}$$

Tatsächliche Kosten für Zugriff auf s_i in $L_{A,i-1}$ mit Algorithmus A: Mindestens $k_i - x_i$, denn $k_i - x_i - 1$ Elemente stehen in beiden Listen vor dem Element s_i, also auch in der Liste $L_{A,i-1}$ vor s_i.

Damit erhalten wir:

$$\sum_{i=1}^{m} a_i \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Übersicht

Wir benötigen einige Vorarbeiten, um die Laufzeiten der einzelnen Operationen eines Fibonacci-Heaps abschätzen zu können.

Lemma 1: Sei x ein Knoten im Fibonacci-Heap, seien $y_1, \ldots, y_k$ die Kinder von x in der zeitlichen Reihenfolge, in der sie an x angehängt wurden. Dann gilt:

$$deg(y_1) \geq 0 \quad \text{und} \quad deg(y_i) \geq i - 2 \text{ für } i = 2, 3, \ldots, k$$

Induktionsanfang: $deg(y_1) \geq 0$ ist klar.

Induktionsschluss für $i \geq 2$:

- Als y_i an x angehängt wurde, waren $y_1, \ldots, y_{i-1}$ bereits Kinder von x, also gilt $deg(x) \geq i - 1$.
- Da y_i an x angehängt wird, gilt: $deg(x) = deg(y_i)$, also gilt $deg(y_i) \geq i - 1$.
- Seither hat Knoten y_i höchstens ein Kind verloren, sonst hätten wir y_i von x abgetrennt, also gilt: $deg(y_i) \geq i - 2$

Die Fibonacci-Zahlen F_i sind wie folgt rekursiv definiert:

$$F_0 = 0, \quad F_1 = 1, \quad F_i = F_{i-1} + F_{i-2} \text{ für } i \geq 2$$

Lemma 2:

$$F_{k+2} \;=\; 1 + \sum_{i=0}^{k} F_i$$

Beweis durch Induktion über k:
Induktionsanfang: $k = 0$

$$F_2 = 1 \;\stackrel{!}{=}\; 1 + \sum_{i=0}^{0} F_i = 1 + 0 = 1$$

Induktionsschluss: $k - 1 \to k$

$$F_{k+2} \;\stackrel{\text{Def.}}{=}\; F_k + F_{k+1} \;\stackrel{\text{I.V.}}{=}\; F_k + (1 + \sum_{i=0}^{k-1} F_i) \;=\; 1 + \sum_{i=0}^{k} F_i$$

oder anschaulich:

$$
\begin{aligned}
F_{k+2} &= F_k + F_{k+1} \\
&= F_k + F_{k-1} + F_k \\
&= F_k + F_{k-1} + F_{k-2} + F_{k-1} \\
&= F_k + F_{k-1} + F_{k-2} + F_{k-3} + F_{k-2} \\
&\vdots \\
&= F_k + F_{k-1} + F_{k-2} + F_{k-3} + \ldots + F_1 + F_2 \\
&= \underbrace{F_k + F_{k-1} + F_{k-2} + F_{k-3} + \ldots + F_1 + F_0}_{=\sum_{i=0}^{k} F_i} + \underbrace{F_1}_{=1} \\
&= \sum_{i=0}^{k} F_i + 1
\end{aligned}
$$

Lemma 3: Sei x ein Knoten in einem Fibonacci-Heap, sei $k = deg(x)$. Dann gilt:

$$size(x) \;\geq\; F_{k+2}$$

Beweis: Bezeichne s_k den minimalen Wert von $size(z)$ über alle Knoten z mit $deg(z) = k$. Dann gilt offensichtlich:

- $s_0 = 1, \;\; s_1 = 2, \;\; s_2 = 3$
- $s_k \leq size(x)$

Da x den Grad k hat, gilt für jedes Kind y_i, $i = 2, \ldots, k$ von x nach Lemma 1: $deg(y_i) \geq i - 2$. Außerdem müssen wir bei $size(x)$ den Knoten x und das Kind y_1 berücksichtigen. Daher erhalten wir:

$$size(x) \;\geq\; s_k \;\geq\; 2 + \sum_{i=2}^{k} s_{i-2} \;=\; 2 + \sum_{i=0}^{k-2} s_i$$

Ferner können wir feststellen

$$\begin{aligned} s_0 = 1 &\geq F_2 = 1 \\ s_1 = 2 &\geq F_3 = 2 \\ s_2 = 3 &\geq F_4 = 3 \end{aligned}$$

und mittels vollständiger Induktion und Lemma 2 zeigen:

$$s_k \;\geq\; 2 + \sum_{i=0}^{k-2} s_i \;\geq\; 2 + \sum_{i=0}^{k-2} F_{i+2} \;=\; 1 + \sum_{i=0}^{k} F_i \;=\; F_{k+2}$$

Insgesamt erhalten wir also:

$$size(x) \;\geq\; s_k \;\geq\; F_{k+2}$$

Aus der Mathematik wissen wir:

$$F_k = \frac{1}{\sqrt{5}} \cdot \phi^k \quad \text{mit} \quad \phi = \frac{(1+\sqrt{5})}{2}$$

ϕ ist der goldene Schnitt mit $\phi \approx 1.61803$.

$\rightarrow$ *Die Fibonacci-Zahlen wachsen exponentiell!*

Korollar: Jeder Knoten x eines Fibonacci-Heaps der Größe n hat höchstens Grad $\mathcal{O}(\log(n))$.

Beweis: Sei $k = deg(x)$, dann gilt nach Lemma 3:

$$n \geq size(x) \geq \phi^k \iff \log_\phi(n) \geq k$$

Potenzial-Funktion: Definiere zu einem Fibonacci-Heap H

$$\Phi(H) \ := \ b(H) + 2 \cdot m(H),$$

wobei

- $b(H)$ die Anzahl der Bäume in der Wurzelliste und
- $m(H)$ die Anzahl der markierten Knoten bezeichnet.

Zur Erinnerung:
Die amortisierten Kosten a_i sind die tatsächlichen Kosten t_i plus der Differenz der Potenziale:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

MakeHeap() $\rightarrow b(H) = 0,\ m(H) = 0$
amortisierte Kosten = aktuelle Kosten $\in \mathcal{O}(1)$

Minimum(H): $\rightarrow b(H)$ und $m(H)$ unverändert
amortisierte Kosten = aktuelle Kosten $\in \mathcal{O}(1)$

Union(H_1, H_2): $\rightarrow \Phi(H) = \Phi(H_1) + \Phi(H_2)$
amortisierte Kosten = aktuelle Kosten $\in \mathcal{O}(1)$

Insert(H, x): $\rightarrow b(H') = b(H) + 1,\ m(H)$ unverändert
amortisierte Kosten = aktuelle Kosten $+ 1 \in \mathcal{O}(1)$

ExtractMin(H):

- Der Knoten, der entfernt wird, hat maximal $\mathcal{O}(\log(n))$ Kinder, die in die Wurzelliste aufgenommen werden.
- Die Wurzelliste wird einmal durchlaufen, um ggf. Bäume gleichen Ranges zu verschmelzen.

$\rightarrow$ aktuelle Kosten: $t_i = b(H) + \mathcal{O}(\log(n))$

- Nach dem Verschmelzen können nur maximal $\mathcal{O}(\log(n))$ viele Bäume in der Wurzelliste stehen, da dann alle Bäume unterschiedlichen Rang haben.
- Es werden keine Knoten markiert.

$\rightarrow$ $\Phi_{i-1} = b(H) + 2m(H)$ und $\Phi_i \leq \mathcal{O}(\log(n)) + 2m(H)$

$\Rightarrow$ amortisierte Kosten: $t_i + \Phi_i - \Phi_{i-1} \in \mathcal{O}(\log(n))$

DecreaseKey(H, x, k): Bei insgesamt c Abtrennungen

- entsteht ein tatsächlicher Aufwand von c Einheiten,
- werden mindestens $c - 1$ Markierungen gelöscht und
- c viele Bäume werden in die Wurzelliste aufgenommen.
- Mit $\Phi_{i-1} = b(H) + 2m(H)$ gilt also:

  $\Phi_i = b(H) + c \ + \ 2(m(H) - (c - 1))$

$\Rightarrow$ amortisierte Kosten: $t_i + \Phi_i - \Phi_{i-1} \in \mathcal{O}(1)$

Delete(H, x):

- Laufzeit ergibt sich aus DecreaseKey und ExtractMin

$\Rightarrow$ amortisierte Kosten $\in \mathcal{O}(\log(n))$

Übersicht

Splay-Bäume im Vergleich zu AVL- und Rot-Schwarz-Bäumen:

- Es sind keine zusätzlichen Informationen wie Balance-Wert oder Farbe notwendig.
  - speicher-effizienter
  - weniger Programmieraufwand
- Strukturanpassung an unterschiedliche Zugriffshäufigkeiten:
  - Oft angefragte Schlüssel werden in Richtung Wurzel bewegt.
  - Selten angefragte Schlüssel wandern zu den Blättern hinab.
- Die Zugriffshäufigkeiten sind vorher nicht bekannt.

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

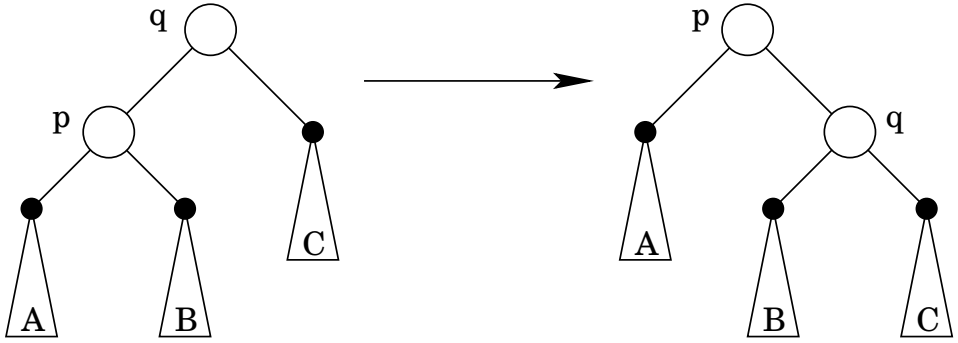
Die Splay-Operation: Sei `T` ein Suchbaum und `x` ein Schlüssel. Dann ist `splay(T,x)` der Suchbaum, den man wie folgt erhält:

- 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 Vorgänger 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.

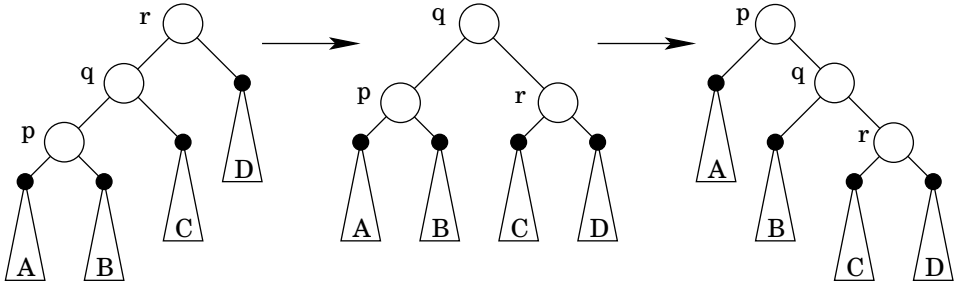
Fall 1: p hat den Vorgänger q und q ist die Wurzel.

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



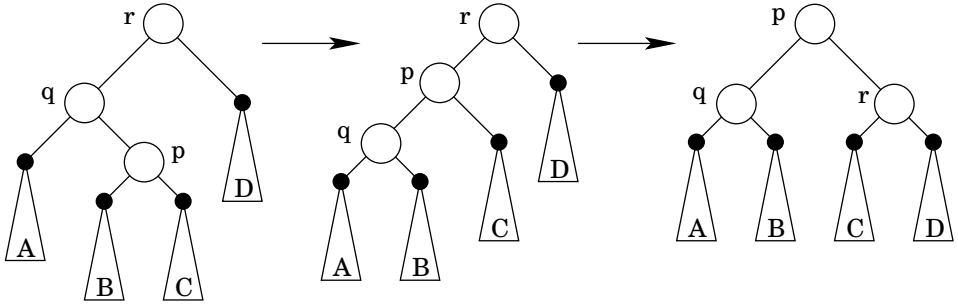
Fall 2: p hat den Vorgänger q und den Vor-Vorgänger r. Außerdem sind p und q beides rechte oder beides linke Nachfolger.

Dann führe die Operation zig-zig aus: zwei aufeinanderfolgende Rotationen in dieselbe Richtung, die p zwei Ebenen hinaufbewegen.



Fall 3: p hat Vorgänger q und Vor-Vorgänger r, wobei p linker Nachfolger von q und q rechter Nachfolger von r ist, bzw. p ist rechter Nachfolger von q und q ist linker Nachfolger von r.

Dann führe die Operation zig-zag aus: zwei Rotationen in entgegengesetzter Richtung, die p zwei Ebenen hinaufbewegen.



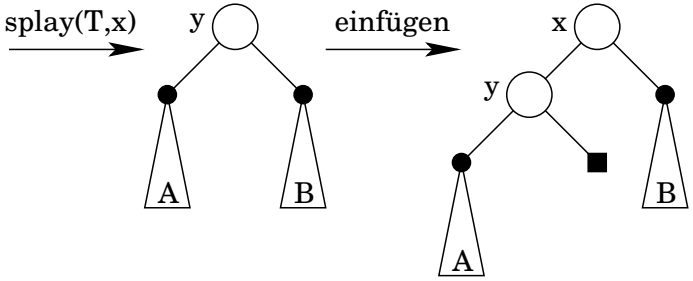
Anmerkungen:

- Kommt `x` in `T` vor, so erzeugt `splay(T,x)` einen Suchbaum, der den 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.

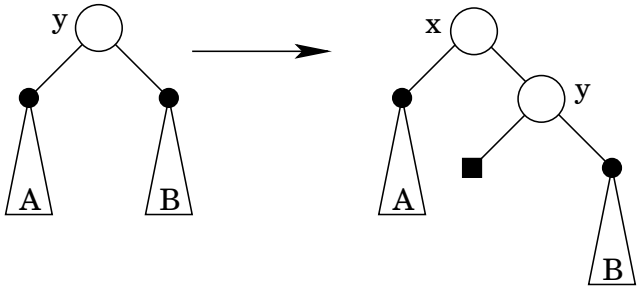
Einfügen: Um `x` in `T` einzufügen, rufe `splay(T,x)` auf. Ist `x` nicht in der Wurzel, so füge wie folgt eine neue Wurzel mit `x` ein.

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



Dieses Vorgehen ist korrekt aufgrund obiger zweiter Anmerkung: 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.

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



Dieses Vorgehen ist korrekt aufgrund obiger zweiter Anmerkung.

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 $\mathcal{O}(m \cdot \log(n))$ Schritte, siehe Kapitel amortisierte Laufzeitanalyse.

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

Ist `x` in der Wurzel, dann sei `L` der linke und `R` der rechte Teilbaum unter der Wurzel.

Rufe `splay(L,` ∞`)` auf. Dadurch entsteht ein Teilbaum mit dem größten Schlüssel von `L` an der Wurzel und einem leeren rechten Teilbaum. Ersetze den rechten leeren Teilbaum durch `R`.

Hinweis: Die Ausführung der Operation `splay(T,x)` schließt immer die Suche nach `x` ein.

Bei Splay-Bäumen werden alle drei Operationen Suchen, Einfügen und Entfernen auf die Splay-Operation zurückgeführt.

Die Kosten einer Splay-Operation sei die Anzahl der ausgeführten Rotationen (plus 1, falls keine Operation ausgeführt wird).

- Jede zig-Operation zählt eine Rotation.
- Jede zig-zig- und zig-zag-Operation zählt zwei Rotationen.

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 $r(p)$.

Die Potenzialfunktion $\Phi(T)$ für einen Suchbaum T sei definiert als 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 $s_v(p)$ und den Rang $r_v(p)$ von p unmittelbar vor Ausführung der Rotation und
- die Größe $s_n(p)$ und den Rang $r_n(p)$ von p unmittelbar nach Ausführung der Rotation.

Wir werden zeigen, dass

- jede zig-zig- und zig-zag-Operation auf p mit amortisiertem Aufwand $3(r_n(p) - r_v(p))$ und
- jede zig-Operation mit amortisiertem Aufwand $3(r_n(p) - r_v(p)) + 1$ 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.

Beachte: Nur die letzte Operation kann eine zig-Operation sein.

Dann ergibt sich als amortisierter Aufwand von $splay(T, x)$:

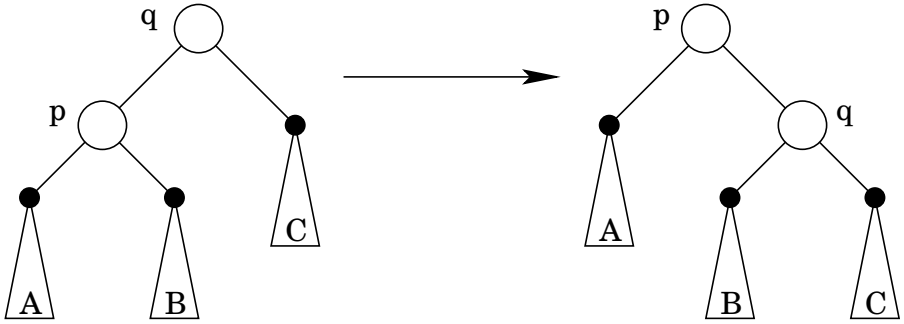
$$\begin{aligned} splay(T,x) &= 3(r^{(1)}(x) - r(x)) \\ &+ 3(r^{(2)}(x) - r^{(1)}(x)) \\ &\vdots \\ &+ 3(r^{(k)}(x) - r^{(k-1)}(x)) + 1 \\ &= 3(r^{(k)}(x) - r(x)) + 1 \end{aligned}$$

Da x durch die k Operationen zur Wurzel gewandert ist, gilt $r^{(k)}(x) = r(T)$ und damit das Lemma.

Noch zu zeigen:

- Jede zig-zig- und zig-zag-Operation auf p ist mit amortisiertem Aufwand $3(r_n(p) - r_v(p))$ und
- jede zig-Operation mit amortisiertem Aufwand $3(r_n(p) - r_v(p)) + 1$ ausführbar.

Fall 1: (zig-Operation)



Dann ist q 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:

$$\begin{aligned}
a_{zig} &= 1 + \Phi_n(T) - \Phi_v(T) \\
&= 1 + (r_n(p) + r_n(q) + \ldots) - (r_v(p) + r_v(q) + \ldots) \\
&= 1 + (r_n(p) + r_n(q)) - (r_v(p) + r_v(q)) \\
&= 1 + r_n(q) - r_v(p), \quad \text{weil } r_n(p) = r_v(q) \\
&\leq 1 + r_n(p) - r_v(p), \quad \text{weil } r_n(p) \geq r_n(q) \\
&\leq 1 + 3(r_n(p) - r_v(p)), \quad \text{weil } r_n(p) \geq r_v(p)
\end{aligned}$$

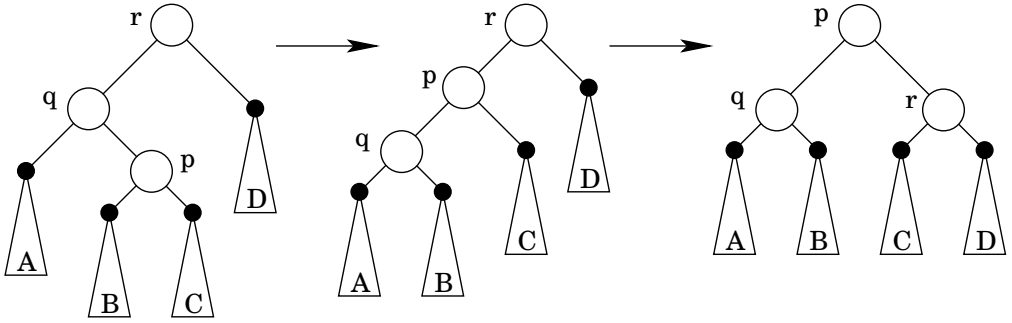
Für die nächsten beiden Fälle, formulieren wir folgende Hilfsaussage:

> Sind a und b positive Zahlen und gilt $a + b \leq c$, so folgt $\log(a) + \log(b) \leq 2\log(c) - 2$.

Da das geometrische Mittel zweier positiver Zahlen niemals größer ist als das arithmetische Mittel, gilt:

$$\begin{aligned}
\sqrt{ab} &\leq (a+b)/2 \\
\sqrt{ab} &\leq c/2 \\
ab &\leq (c/2)(c/2) \\
\log(ab) &\leq \log((c/2)(c/2)) \\
\log(a) + \log(b) &\leq 2\log(c/2) \\
\log(a) + \log(b) &\leq 2\log(c) - 2\log(2) \\
\log(a) + \log(b) &\leq 2\log(c) - 2
\end{aligned}$$

Fall 2: (zig-zag-Operation)



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 $r_n(p) = r_v(r)$.

Also gilt für die amortisierten Kosten:

$$\begin{aligned} a_{zig-zag} &= 2 + \Phi_n(T) - \Phi_v(T) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r) + \ldots) \\ &\quad - (r_v(p) + r_v(q) + r_v(r) + \ldots) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r)) - (r_v(p) + r_v(q) + r_v(r)) \\ &= 2 + r_n(q) + r_n(r) - r_v(p) - r_v(q), \quad \text{weil } r_n(p) = r_v(r) \end{aligned}$$

Weil p vor Ausführung der zig-zag-Operation Kind von q war, gilt $r_v(q) \geq r_v(p)$ und somit

$$a_{zig-zag} \leq 2 + r_n(q) + r_n(r) - 2 \cdot r_v(p)$$

Wir hatten zuletzt festgestellt:

$$a_{zig-zag} \leq 2 + r_n(q) + r_n(r) - 2 \cdot r_v(p)$$

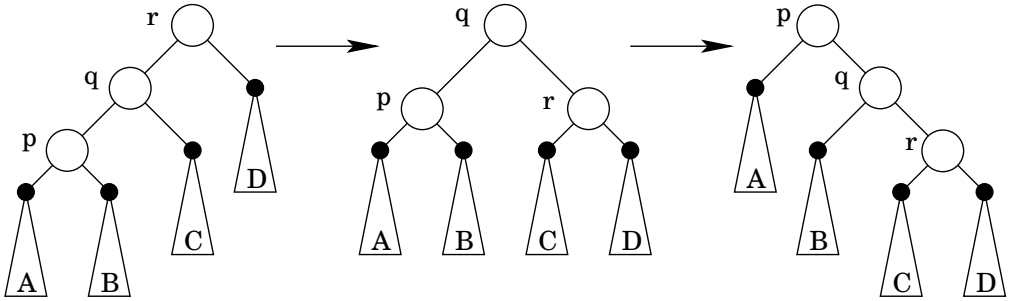
Ferner gilt $s_n(q) + s_n(r) \leq s_n(p)$ und damit aufgrund der Definition des Ranges und obiger Hilfsaussage:

$$\begin{aligned} &\log(s_n(q)) + \log(s_n(r)) \leq 2 \cdot \log(s_n(p)) - 2 \\ \iff \quad &r_n(q) + r_n(r) \leq 2 \cdot r_n(p) - 2 \end{aligned}$$

Damit erhalten wir schließlich:

$$\begin{aligned} a_{zig-zag} &\leq 2 + r_n(q) + r_n(r) - 2 \cdot r_v(p) \\ &\leq 2 + (2 \cdot r_n(p) - 2) - 2 \cdot r_v(p) = 2(r_n(p) - r_v(p)) \\ &\leq 3(r_n(p) - r_v(p)), \quad \text{weil } r_n(p) \geq r_v(p) \end{aligned}$$

Fall 3: (zig-zig-Operation)



Die Operation hat tatsächliche Kosten von 2, weil zwei Rotationen durchgeführt werden.

Auch hier können sich höchstens die Ränge von p, q und r ändern, und es gilt wie im vorherigen Fall $r_n(p) = r_v(r)$.

Für die amortisierten Kosten gilt:

$$\begin{aligned} a_{zig-zig} &= 2 + \Phi_n(T) - \Phi_v(T) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r) + \ldots) \\ &\quad - (r_v(p) + r_v(q) + r_v(r) + \ldots) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r)) - (r_v(p) + r_v(q) + r_v(r)) \\ &= 2 + r_n(q) + r_n(r) - r_v(p) - r_v(q), \quad \text{weil } r_n(p) = r_v(r) \end{aligned}$$

Da vor Ausführung der zig-zig-Operation p Kind von q und nachher q Kind von p ist, folgt $r_v(p) \leq r_v(q)$ und $r_n(p) \geq r_n(q)$, und somit

$$a_{zig-zig} \leq 2 + r_n(p) + r_n(r) - 2 \cdot r_v(p)$$

Zuletzt hatten wir festgestellt:

$$a_{zig-zig} \leq 2 + r_n(p) + r_n(r) - 2 \cdot r_v(p)$$

Diese Summe ist genau dann kleiner oder gleich $3(r_n(p) - r_v(p))$, wenn $r_v(p) + r_n(r) \leq 2 \cdot r_n(p) - 2$ gilt.

$$\begin{aligned} 2 + \cancel{r_n(p)} + r_n(r) - \cancel{2r_v(p)} &\overset{!}{\leq} \cancel{r_n(p)} - r_v(p) + 2r_n(p) - \cancel{2r_v(p)} \\ 2 + r_n(r) + r_v(p) &\overset{!}{\leq} 2r_n(p) \\ r_n(r) + r_v(p) &\overset{!}{\leq} 2r_n(p) - 2 \end{aligned}$$

Aus der zig-zig-Operation folgt, dass $s_v(p) + s_n(r) \leq s_n(p)$. Mit obiger Hilfsaussage und der Definition der Ränge erhält man die obige Ungleichung.

Damit ist das Lemma bewiesen.

Satz: Das Ausführen einer beliebigen Folge von m Such-, Einfüge- und Entferne-Operationen, in der höchstens n-mal Einfügen vorkommt und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $\mathcal{O}(m \cdot \log(n))$ 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.

Übersicht

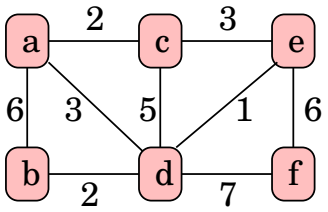
Sei $G = (V, E, c)$ ein ungerichteter Graph mit Kostenfunktion $c : E \to \mathbb{R}^+$.

```
A := ∅
for each vertex v ∈ V do
    create(v)
sort the edges of E by non-decreasing weight
for each edge (u, v) ∈ E do
    r_u := find(u)
    r_v := find(v)
    if r_u ≠ r_v then
        A := A ∪ {{u, v}}
        union(r_u, r_v)
```

Nach Ablauf des Algorithmus enthält die Menge A die Kanten eines minimalen Spannbaums.

Zunächst werden die Kanten anhand ihrer Gewichtung sortiert. Anschließend wird in jedem Schritt eine Kante $e = \{u, v\}$ aus der sortierten Liste entfernt und geprüft, ob die Endknoten u und v in verschiedenen Mengen liegen. Falls ja, dann gehört e zum minimalen Spannbaum und die entsprechenden Mengen werden verschmolzen.

Beispiel:



sorted(E)	Knotenmengen
initial	$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$
$\{d, e\}$	$\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}$
$\{a, c\}$	$\{a, c\}, \{b\}, \{d, e\}, \{f\}$
$\{b, d\}$	$\{a, c\}, \{b, d, e\}, \{f\}$
$\{a, d\}$	$\{a, c, b, d, e\}, \{f\}$
$\{c, e\}$	unverändert
$\{c, d\}$	unverändert
$\{a, b\}$	unverändert
$\{e, f\}$	$\{a, c, b, d, e, f\}$
$\{d, f\}$	unverändert

Wir nutzen eine Datenstruktur, mit der wir effizient disjunkte Mengen verwalten können. Insbesondere werden die Funktionen `find` und `union` gut unterstützt.

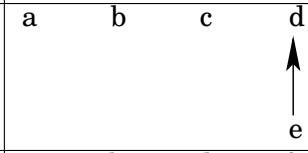
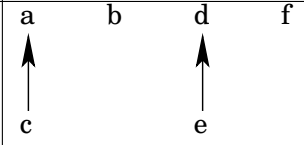
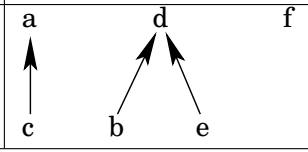
Union-Find-Datenstruktur:

- Zur Speicherung einer disjunkten Zerlegung einer Menge

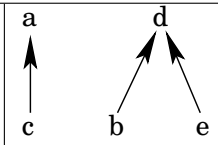
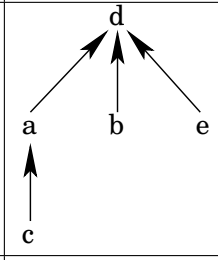
$$S = X_1 \cup X_2 \cup \ldots \cup X_k \text{ mit } X_i \cap X_j = \emptyset \text{ für } i \neq j.$$

- Speichere jede Klasse X_i in einem Baum.
- Der Repräsentant einer Klasse X_i ist die Wurzel des Baums.
- Die Funktion `find(`v`)` liefert den Repräsentanten des Baums, in dem Knoten v gespeichert ist.
- Damit ein Knoten schnell gefunden werden kann, werden Zeiger auf alle Elemente $v \in S$ gespeichert.
- Die Funktion `union` hängt den flacheren Baum an die Wurzel des tieferen Baums an. $\rightarrow$ Vereinigung nach Höhe

In unserem Beispiel:

sorted(E)	Mengenrepräsentation durch Bäume
initial	a b c d e f
$\{d, e\}$	
$\{a, c\}$	
$\{b, d\}$	

Fortsetzung:

sorted(E)	Mengenrepräsentation durch Bäume
$\{b, d\}$	
$\{a, d\}$	
...	...

Die Funktion `create(`v`)` initialisiert den Vorgänger und den Rang der Wurzel:

$$\begin{array}{|l} \text{pred}[v] := 0 \\ \text{rang}[v] := 0 \end{array}$$

Laufzeit: $\mathcal{O}(1)$

Wir speichern zu jedem Knoten dessen Rang, um eine Vereinigung zweier Bäume nach ihrer Höhe durchführen zu können. Aktualisiert wird nur der Rang der Wurzel bei `union(`r_u`,`r_v`)`:

$$\begin{array}{|lll} \text{wenn} & \text{rang}[r_u] < \text{rang}[r_v] & \\ & \text{dann pred}[r_u] := r_v & \\ \text{sonst} & \text{pred}[r_v] := r_u & \\ & \text{wenn} \quad \text{rang}[r_u] = \text{rang}[r_v] & \\ & \qquad \text{dann rang}[r_u] := \text{rang}[r_u] + 1 & \end{array}$$

Laufzeit: $\mathcal{O}(1)$

Dabei sind r_u und r_v die Repräsentanten der Mengen, die u bzw. v enthalten.

Übung 57. *Zeigen Sie mittels vollständiger Induktion über die Höhe h eines Baums, dass die Höhe der so entstehenden Bäume nur logarithmisch in der Anzahl der Knoten ist. Also: Ein Baum der Höhe h hat mindestens 2^h viele Knoten.*

Folgerung: Die Laufzeit einer `find`-Operation ist logarithmisch in der Anzahl der Knoten beschränkt.

Laufzeit:

- Initialisierung: $\mathcal{O}(\mathcal{V})$
- Sortieren der Kanten: $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{E}))$
- Schleifendurchläufe mal Aufwand `find` und `union`: $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$
- Gesamtlaufzeit: $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$

Übung 58. *Zeigen Sie, dass $\mathcal{O}(\log(\mathcal{E})) = \mathcal{O}(\log(\mathcal{V}))$ gilt.*

Anmerkung: Die Kosten der `find`-Operation sind abhängig von der Höhe der Bäume.

- Wäre es nicht günstiger, alle Knoten direkt an die Wurzel zu hängen?
- Dann hätte die `find`-Operation nur Laufzeit $\mathcal{O}(1)$, aber die `union`-Operation wäre teurer als bisher.

Weighted-Union: Hänge alle Knoten des kleineren Baums (bzgl. der Anzahl Knoten) direkt unter die Wurzel des größeren Baums. Werden zwei Mengen A und B vereinigt, ergäbe sich als Laufzeit für die `union`-Operation: $\mathcal{O}(\min\{|A|, |B|\})$

Für eine solche Implementierung ergäbe sich die folgende amortisierte Laufzeit:

- Nach n vielen `create`-Operationen kann es maximal $n - 1$ viele `union`-Operationen geben.
- Man kann zeigen: Eine Folge von m Operationen, unter denen maximal n viele `create`-Operationen sind, hat die amortisierte Laufzeit von $\mathcal{O}(m + n \cdot \log(n))$.

Laufzeit von Kruskals Algorithmus bei Weighted-Union:

- Die Laufzeit der Schleife bei Kruskals Algorithmus würde sich von $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$ auf $\mathcal{O}(\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))$ verringern.
- Aufgrund des initialen Sortierens der Kanten wäre die gesamte Laufzeit aber immer noch $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$.

Entfällt das initiale Sortieren der Kanten, oder lassen sich die Kanten in linearer Zeit sortieren (z.B. ganze Zahlen mittels Radix-Sort), dann ergäbe sich $\mathcal{O}(\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))$ als Laufzeit von Kruskals Algorithmus bei Verwendung der Weighted-Union-Strategie.

Wie wir mittels einer Modifikation der ursprünglichen Union-Find-Struktur eine noch bessere Laufzeit erhalten, schauen wir uns jetzt an.

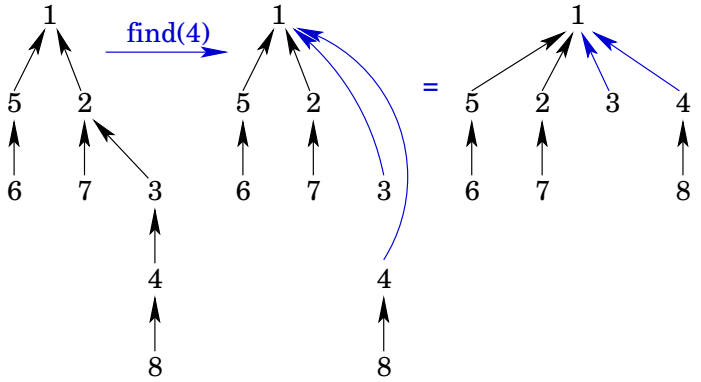
Eine bessere Laufzeit erhält man mit folgender Idee:

- `union`-Operation: Hänge die Wurzel des flacheren Baums an die Wurzel des höheren Baums, also Union-by-rank.
- Verkürze während der `find`-Operation die Pfadlängen.

`find(`x`)` mit Pfadkomprimierung:

```
res := x
z := x
while pred[res] ≠ 0 do
        res = pred[res]
while pred[z] ≠ res do
        tmp := z
        z := pred[z]
        pred[tmp] := res
return res
```

Beispiel:



Die Pfadkomprimierung macht die `find`-Methode ungefähr doppelt so teuer wie vorher, die asymptotische Laufzeit wird nicht größer.

Nachfolgende `find`-Operationen werden aber preiswerter!

Auf den nächsten Folien schauen wir uns dazu die amortisierte Laufzeitanalyse an.

Betrachten wir noch einmal die Struktur und die einzelnen Operationen der Union-Find-Datenstruktur:

```
struct Node {
    int rank;              // depth of tree
    Node *parent;          // reversed tree
};

void create(Node *x) {
    x->parent = x;
    x->rank = 0;
}

Node *find(Node *x) {  // path compression
    if (x->parent != x)
        x->parent = find(x->parent);
    // rank wird nicht aktualisiert !
    return x->parent;
}
```

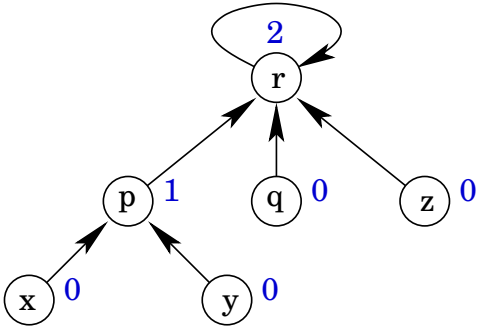
```
// union by rank  (union ist Schlüsselwort in C/C++)
Node *mergeSets(Node *x, Node *y) {
    if (x->rank > y->rank) {
        y->parent = x;
        return x;
    }

    x->parent = y;
    if (x->rank == y->rank)
        y->rank += 1;
    return y;
}
```

Im Folgenden:

- Die Wurzel des Union-Find-Baums nennen wir Repräsentanten (leader) der Menge.
- Wir betrachten beliebige Folgen von m vielen Operationen `union`, `find` und `create` über insgesamt n Elementen, es gab also n `create`-Operationen.

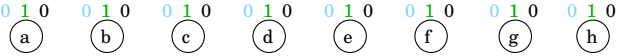
Wir behalten die Sprechweise bzgl. Kinder und Vorgänger bei, obwohl die Bäume von den Blättern zur Wurzel gerichtet sind.



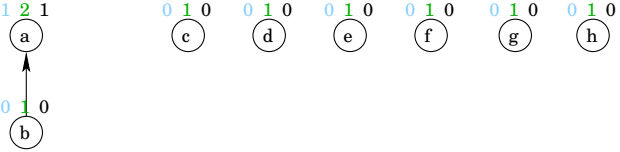
- r ist die Wurzel des Baums.
- p ist der Vorgänger von x und y.
- r ist der Repräsentant der Menge $\{p, q, r, x, y, z\}$.
- p ist die Wurzel des Teilbaums mit den Elementen $\{p, x, y\}$.

Beispiel:

u.height u.size u.rank

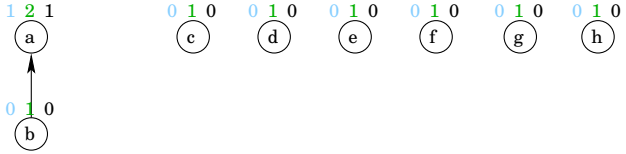


`union(a,b)`

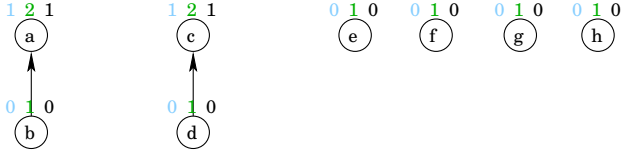


Union-Find-Datenstruktur

u.height u.size u.rank

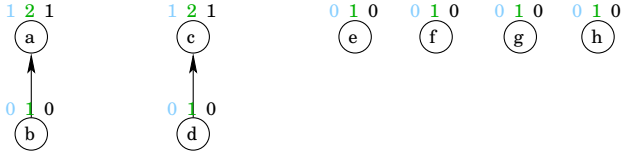


`union(c,d)`

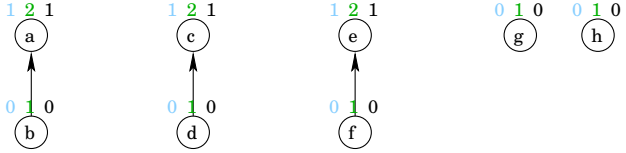


Union-Find-Datenstruktur

u.height u.size u.rank

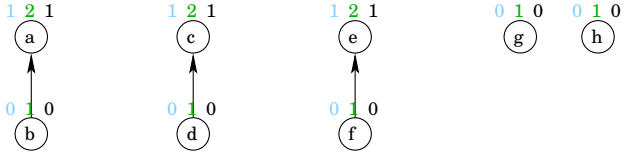


`union(e,f)`

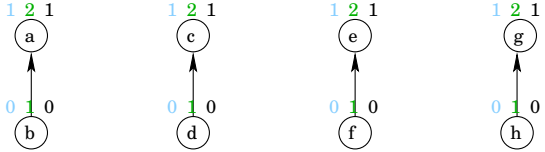


Union-Find-Datenstruktur

u.height u.size u.rank

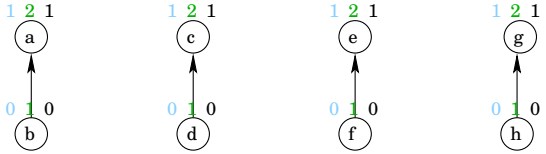


`union(g,h)`

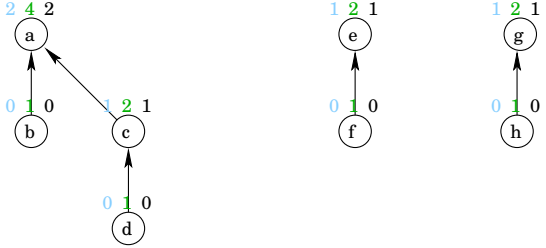


Union-Find-Datenstruktur

u.height u.size u.rank

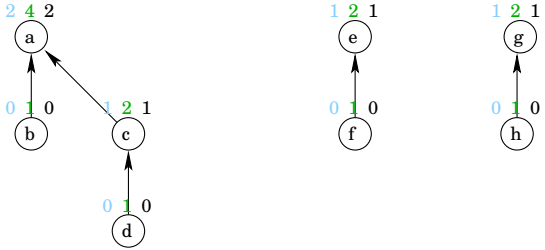


`union(a,c)`

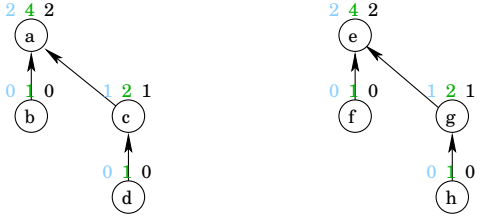


Union-Find-Datenstruktur

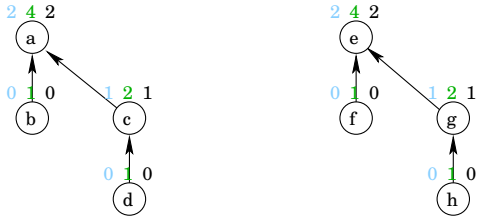
u.height u.size u.rank



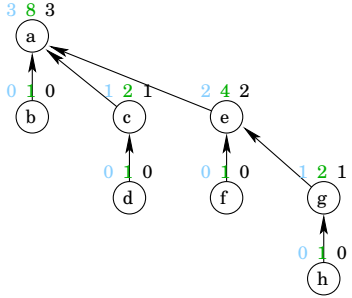
`union(e,g)`



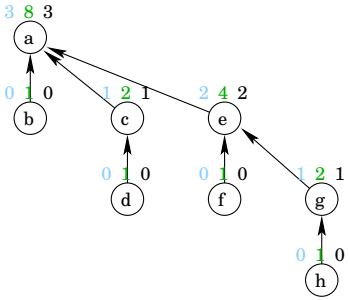
u.height u.size u.rank



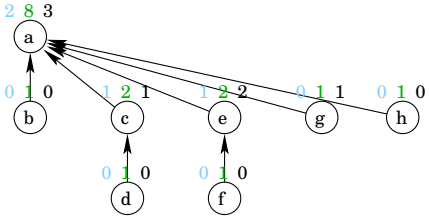
`union(a,e)`



u.height u.size u.rank



`find(h)` durch Pfadkomprimierung wird der Rang nicht aktualisiert



Sammeln wir zunächst einige Eigenschaften:

- Nur die Ränge von Repräsentanten werden geändert. Wenn ein Knoten kein Repräsentant einer Menge ist, dann wird sein Rang nicht mehr verändert.
  - Frage: Warum aktualisieren wir während der find-Operation nicht die Ränge der Knoten?
- Ist ein Knoten kein Repräsentant einer Menge, dann ist sein Rang echt kleiner als der Rang seines Vorgängers.
  - anders gesagt: Die Werte der Ränge auf einem Weg von einem Knoten zur Wurzel sind streng monoton steigend.
- Es gibt höchstens $\left\lfloor \frac{n}{2^r} \right\rfloor$ Knoten mit Rang r.

  Beweisidee: Die Pfadkomprimierung ändert die Ränge nicht.
  - I.A.: Es gibt höchstens $\lfloor \frac{n}{2^0} \rfloor = n$ Knoten vom Rang 0.
  - I.S.: Um einen Knoten vom Rang r zu erzeugen, werden zwei Knoten von Rang $r-1$ benötigt. Genau einer der beiden bekommt den Rang r.
    Also gibt es höchstens $\lfloor \frac{n}{2^{r-1}} \rfloor \cdot \frac{1}{2} = \lfloor \frac{n}{2^r} \rfloor$ viele Knoten vom Rang r.

Definition:

$$\text{tower}(b) := \begin{cases} 1 & \text{falls } b = 0 \\ 2^{\text{tower}(b-1)} & \text{für } b \geq 1 \end{cases}$$

anschaulich:

$$\text{tower}(b) := \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\} b$$

einige Werte:

b	0	1	2	3	4	5	$\cdots$
$\text{tower}(b)$	1	$2^1 = 2$	$2^2 = 4$	$2^4 = 16$	$2^{16} = 65536$	2^{65536}	$\cdots$

zur Information: Die Zahl 2^{65536} hat 19.729 Dezimalstellen.

Definition:

$$\log^{(i)}(n) := \begin{cases} n & \text{falls } i = 0 \\ \log(\log^{(i-1)}(n)) & \text{für } i \geq 1 \end{cases}$$

Wir definieren $\log^\star(n) := \min\{i \geq 0 \mid \log^{(i)}(n) < 2\}$.

Anders gesagt: Wie oft muss man die log-Taste des Taschenrechners nach Eingabe von n drücken, bis ein Wert kleiner als 2 angezeigt wird.

- $\log^\star(2^1) = 1$
- $\log^\star(2^2) = 2$
- $\log^\star(2^{2^2}) = \log^\star(2^4) = 3$
- $\log^\star(2^{2^{2^2}}) = \log^\star(2^{16}) = 4$
- $\log^\star(2^{2^{2^{2^2}}}) = \log^\star(2^{65536}) = 5$

Die Funktion $\log^\star$ ist die Umkehrfunktion zu tower, also gilt: $\log^\star(\text{tower}(b)) = b$

Zur Laufzeitabschätzung wollen wir den Knoten Blöcke zuordnen. Ein Knoten x wird dem Block b zugeordnet, wenn gilt:

$$\text{tower}(b-1) \;<\; \mathit{rank}(x) \;\leq\; \text{tower}(b)$$

Damit erhalten wir die folgenden Blöcke:

$$\begin{array}{lllll}
b=1: & \text{tower}(0)=1 & < & \mathit{rank}(x) & \leq \quad \text{tower}(1)=2 \\
b=2: & \text{tower}(1)=2 & < & \mathit{rank}(x) & \leq \quad \text{tower}(2)=4 \\
b=3: & \text{tower}(2)=4 & < & \mathit{rank}(x) & \leq \quad \text{tower}(3)=16 \\
b=4: & \text{tower}(3)=16 & < & \mathit{rank}(x) & \leq \quad \text{tower}(4)=65536 \\
b=5: & \text{tower}(4)=65536 & < & \mathit{rank}(x) & \leq \quad \text{tower}(5)=2^{65536}
\end{array}$$

Die Knoten in Block b haben maximal $\text{tower}(b) - \text{tower}(b-1)$ verschiedene Ränge.

Da jeder Knoten einen Rang zwischen 0 und $\log(n)$ hat, gibt es nur die Blocknummern 0 bis $\log^\star(\log(n)) = \log^\star(n) - 1$, also insgesamt höchstens $\log^\star(n)$ Blöcke.

Da es höchstens $\lfloor \frac{n}{2^r} \rfloor$ Elemente vom Rang r gibt, ist die gesamte Anzahl der Knoten in Block b höchstens:

$$\begin{aligned}
\sum_{r=\text{tower}(b-1)+1}^{\text{tower}(b)} \left\lfloor \frac{n}{2^r} \right\rfloor &\leq n \cdot \sum_{r=\text{tower}(b-1)+1}^{\text{tower}(b)} \frac{1}{2^r} \\
&\overset{(*)}{=} n \cdot \left(\frac{1}{2^{\text{tower}(b-1)}} - \frac{1}{2^{\text{tower}(b)}} \right) \\
&= n \cdot \left(\frac{1}{\text{tower}(b)} - \frac{1}{\text{tower}(b+1)} \right) \\
&= \frac{n}{\text{tower}(b)} - \frac{n}{\text{tower}(b+1)} \leq \frac{n}{\text{tower}(b)}
\end{aligned}$$

$(*)$ $\sum_{i=a}^{b} \frac{1}{2^i} = \sum_{i=0}^{b} \frac{1}{2^i} - \sum_{i=0}^{a-1} \frac{1}{2^i} = \frac{1}{2^{a-1}} - \frac{1}{2^b}$ mittels geometrischer Summe

Beispiel: $\frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} = \frac{8}{64} + \frac{4}{64} + \frac{2}{64} + \frac{1}{64} = \frac{16}{64} - \frac{1}{64} = \frac{1}{4} - \frac{1}{64}$

Anzahl Ränge in Block 1 ist höchstens $\text{tower}(1) - \text{tower}(0) = 2 - 1 = 1$
Anzahl Knoten in Block 1 ist höchstens

$$\frac{n}{2^2} \leq \frac{n}{\text{tower}(1)} - \frac{n}{\text{tower}(2)} < \frac{n}{2} = \frac{n}{\text{tower}(1)}$$

Anzahl Ränge in Block 2 ist höchstens $\text{tower}(2) - \text{tower}(1) = 4 - 2 = 2$
Anzahl Knoten in Block 2 ist höchstens

$$\frac{n}{2^3} + \frac{n}{2^4} \leq \frac{n}{\text{tower}(2)} - \frac{n}{\text{tower}(3)} < \frac{n}{2^2} = \frac{n}{\text{tower}(2)}$$

Anzahl Ränge in Block 3 ist höchstens $\text{tower}(3) - \text{tower}(2) = 16 - 4 = 12$
Anzahl Knoten in Block 3 ist höchstens

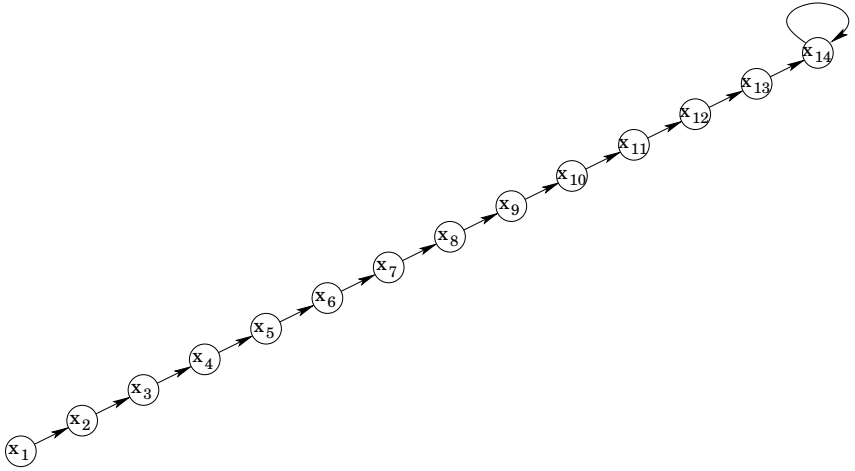
$$\frac{n}{2^5} + \ldots + \frac{n}{2^{16}} \leq \frac{n}{\text{tower}(3)} - \frac{n}{\text{tower}(4)} < \frac{n}{2^{2^2}} = \frac{n}{\text{tower}(3)}$$

Satz: [Tarjan 1977] Wird `find` mit Pfadverkürzung und `union` mit Vereinigung nach Rang implementiert, dann benötigt die Ausführung einer beliebigen Folge von m Operationen, von denen n create-Operationen sind, nur $\mathcal{O}(m \cdot \log^\star(n))$ viele Schritte.

Frage: Was heißt das für die Laufzeit von Kruskals Algorithmus?

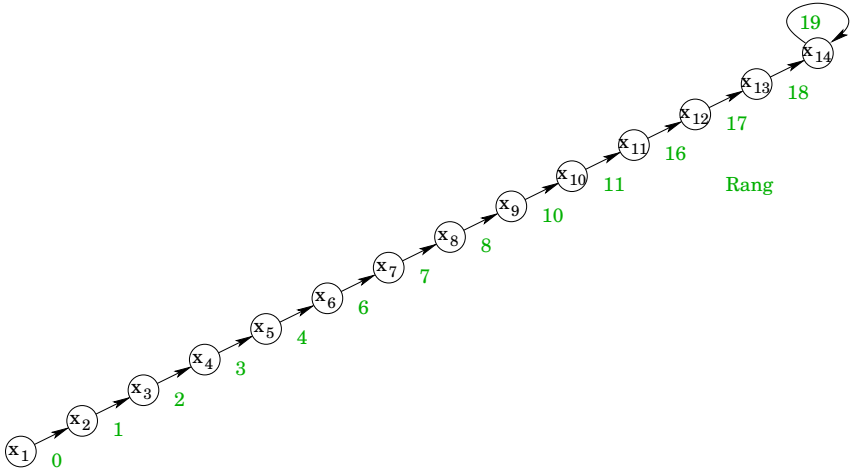
Entfällt das initiale Sortieren der Kanten, oder lassen sich die Kanten in linearer Zeit sortieren (z.B. ganze Zahlen mittels Radix-Sort), dann ergibt sich $\mathcal{O}(\mathcal{E} \cdot \log^\star(\mathcal{V}))$ als Laufzeit, für alle praktischen Eingaben also eine lineare Laufzeit.

Beweis zum Satz von Tarjan: Sei $x_1, x_2, x_3, \ldots, x_\ell$ der Weg vom Knoten x_1 zum Repräsentanten x_ℓ des Baums. Die Kosten von `find`(x_1) entsprechen der Länge ℓ des Weges.

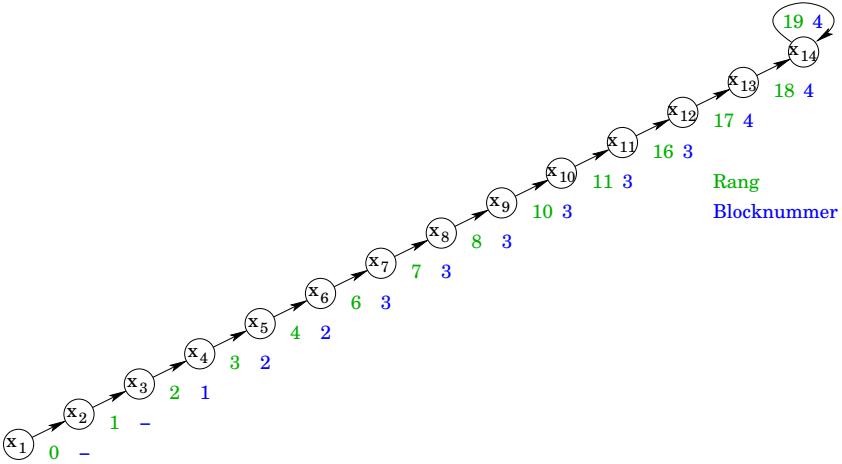


Fortsetzung Beweis: Die Knoten x_i mit $i \geq 3$ haben Rang mindestens 2, da die Ränge bei 0 beginnen und entlang der Vorgänger streng monoton steigen.

Die Ränge der Knoten $x_1, \ldots, x_{\ell-1}$ ändern sich nicht.



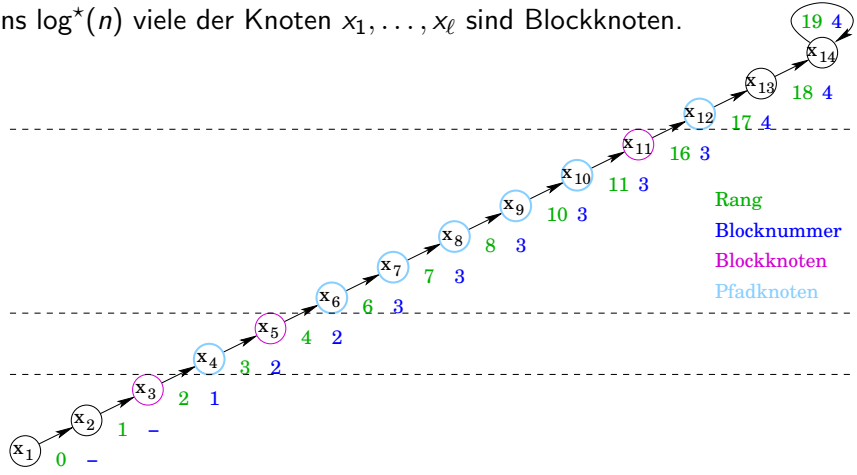
Fortsetzung Beweis: Die Knoten x_i mit $i \geq 3$ haben Blocknummern, die entlang der Vorgänger monoton steigen.



Fortsetzung Beweis: Knoten x_i mit $rang(x_i) \geq 2$

- und $block(x_{i+1}) > block(x_i)$ nennen wir Blockknoten.
- und $block(x_{i+1}) = block(x_i)$ und $i \leq \ell - 2$ nennen wir Pfadknoten.

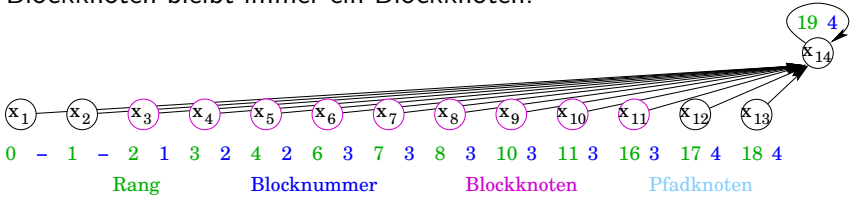
Höchstens $\log^\star(n)$ viele der Knoten $x_1, \ldots, x_\ell$ sind Blockknoten.



Fortsetzung Beweis:

- Bei einer Pfadkomprimierung kann ein Blockknoten einen neuen Vorgänger erhalten. Dieser hat jedoch einen höheren Rang als der alte Vorgänger.
- Bei einer `union`-Operation kann sich der Rang eines Vorgängers nur erhöhen.

→ Ein Blockknoten bleibt immer ein Blockknoten.



- Im Block b gibt es höchstens tower(b) verschiedene Ränge.

→ Jeder Knoten u im Block b mit einem Vorgänger, der nicht die Wurzel ist, wird nach spätestens tower(b) Pfadkomprimierungen, an denen er beteiligt ist, zum Blockknoten[32]. → Aus einem Pfadknoten kann ein Blockknoten werden.

[32] Da sich sein Rang nicht ändert, bleibt er immer in Block b.

Fortsetzung Beweis:

- Jeder Pfadknoten x hat einen Rang ≥ 2 und einen Vorgänger, der nicht die Wurzel des Baumes ist.
- Alle Vorgänger der Pfadknoten werden bei der Pfadkomprimierung durch x_ℓ ersetzt.
- Alle Pfadknoten haben nach der Pfadkomprimierung einen Vorgänger mit einem höheren Rang als die Vorgänger vor der Pfadkomprimierung.

Fortsetzung Beweis:

- Es gibt höchstens $\frac{n}{\text{tower}(b)}$ viele Knoten im Block b.
- Es gibt maximal $\log^\star(n)$ viele Blöcke.
- Jeder Knoten im Block b mit einem Vorgänger, der nicht die Wurzel ist, wird nach spätestens $\text{tower}(b)$ Pfadkomprimierungen, an denen er beteiligt ist, zum Blockknoten.

$\rightarrow$ Bei allen find-Operationen zusammen werden höchstens

$$\sum_{b=1}^{\log^\star(n)} \frac{n}{\text{tower}(b)} \cdot \text{tower}(b) \;=\; \sum_{b=1}^{\log^\star(n)} n \;=\; n \cdot \log^\star(n)$$

viele Pfadknoten besucht.

Fortsetzung Beweis: Die gesamte Laufzeit aller find-Operationen ist daher aus

$$\mathcal{O}(\overbrace{m}^{(1)} \cdot (\overbrace{4}^{(2)} + \overbrace{\log^\star(n)}^{(3)}) + \overbrace{n \cdot \log^\star(n)}^{(4)}) \;=\; \mathcal{O}(m \cdot \log^\star(n)).$$

(1) obere Schranke für die Anzahl der find-Operationen

(2) Knoten x_1, x_2, x_{13} und x_{14}, die weder Pfad- noch Blockknoten sind

(3) maximale Anzahl von Blockknoten

(4) maximale Anzahl insgesamt besuchter Pfadknoten

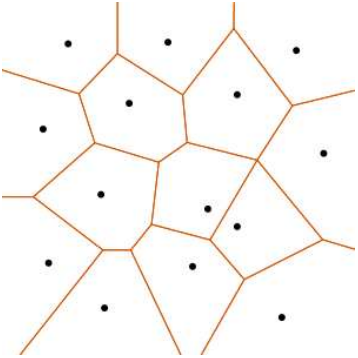
Da jede union-Operation nur eine konstante Laufzeit hat, haben wir damit den Satz von Tarjan bewiesen. □

Übersicht

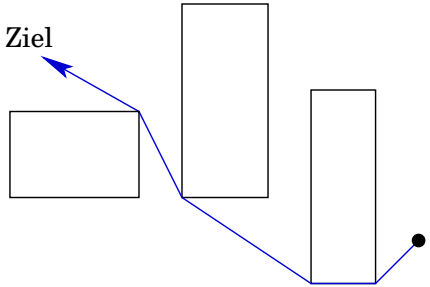
Beispiel: Angenommen, wir kennen die Standorte aller Briefkästen in der Stadt. Welcher Briefkasten ist unserem aktuellen Standort am nächsten?

Um diese Frage beantworten zu können, unterteilen wir die Ebene in sogenannte Voronoi-Regionen:

- Jede Region besitzt ein Zentrum: der Punkt der Punktemenge, der diese Region definiert.
- Eine Region umfasst alle Punkte der Ebene, die näher an dem Zentrum dieser Region liegen, als an irgend einem anderen Zentrum.
- Voronoi-Region: Menge aller Anfragepunkte, für die die obige Antwort dieselbe ist.



Beispiel: Ein Roboter soll einen Brief für uns einwerfen. Wie findet der Roboter sein Ziel, ohne gegen Hindernisse zu laufen?



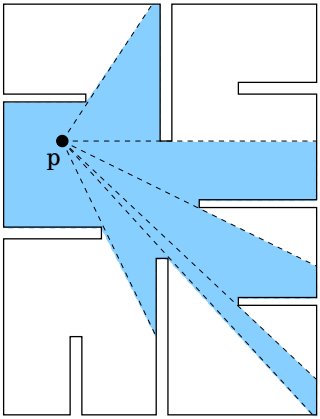
Bewegungsplanung in der Robotik:

- Gegeben: eine Menge von Hindernissen, ein Start- und ein Zielpunkt
- Aufgabe: Finde einen kollisionsfreien kürzesten Weg zum Ziel.

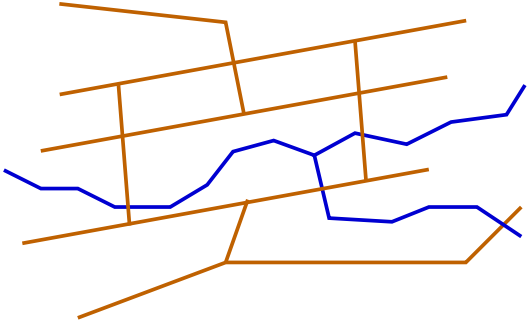
Beispiel: Das Sichtbarkeitspolygon $vis(p)$ eines Punktes p ist ein Objekt des $\mathbb{R}^2$ und ist der Teil eines einfachen Polygons P, der vom Punkt p aus sichtbar ist.

Anwendung zum Beispiel beim Museumsproblem bzw. Art-Gallery-Problem:

Ein Museum, das dargestellt wird durch das Polygon P, soll mit Kameras überwacht werden.
Wie viele Kameras werden benötigt, und wo müssen die Kameras platziert werden, damit jeder Punkt im Inneren von P gesehen wird?



Beispiel: In Geo-Informationssystemen werden Straßen, Flüsse, Kanäle usw. durch Mengen von einzelnen Strecken modelliert und in verschiedenen Ebenen gespeichert. Durch Überlagerung kann man bestimmen, wo Brücken gebaut werden müssen.



Alle Kantenpaare zu testen ist aber viel zu langsam. Wie sehen effiziente Verfahren zur Schnittpunktberechnung aus?

Beispiel: Wie finden wir zu einem gegebenen Anfragepunkt q das Land, in welchem sich der Punkt q befindet?

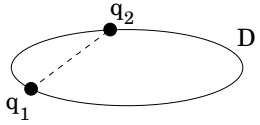
Beispiel: Navigationssysteme müssen zu einem gegebenen Kartenausschnitt effizient alle benötigten Daten bestimmen.

Jedes einzelne Kartenobjekt zu prüfen wäre zu zeitintensiv. Wir benötigen daher eine Datenstruktur zur schnellen Beantwortung von Bereichsanfragen.

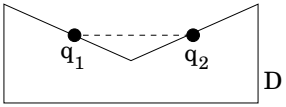
- Ein d-Tupel $(x_1, \ldots, x_d) \in \mathbb{R}^d$ ist ein *Punkt*.
- Für zwei Punkte $p = (p_1, \ldots, p_d)$ und $q = (q_1, \ldots, q_d)$ ist der *euklidische Abstand* definiert als:

$$|pq| = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$$

- Für $q_1, q_2 \in \mathbb{R}^d$ und $\alpha \in \mathbb{R}$
  - ist $q_1 + \alpha \cdot (q_2 - q_1)$ eine *Linie*, und
  - für $0 \leq \alpha \leq 1$ ist $\overline{q_1 q_2}$ das *Liniensegment* $q_1 + \alpha \cdot (q_2 - q_1)$.
- Eine Menge $D \subseteq \mathbb{R}^d$ heißt *konvex*, falls für alle Punktpaare $q_1, q_2 \in D$ gilt: $\overline{q_1 q_2} \subseteq D$



konvex



nicht konvex

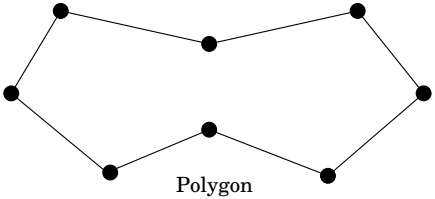
- Eine endliche Menge von Liniensegmenten

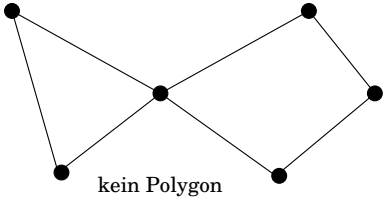
$$P = \{\overline{q_1q_2},\ \overline{q_2q_3},\ \overline{q_3q_4}, \ldots,\ \overline{q_{n-1}q_n}\} \text{ mit } q_i \in \mathbb{R}^2$$

heißt *Polygon*, falls

$$\overline{q_iq_{i+1}} \cap \overline{q_jq_{j+1}} \subseteq \{q_i, q_{i+1}\}, 1 \leq i, j \leq n-1$$

gilt und sich in jedem Punkt maximal 2 Liniensegmente schneiden.





Frage: Wie berechnet man für zwei Liniensegmente $l = (p_1, p_2)$ und $g = (p_3, p_4)$, die jeweils durch ihre Endpunkte $p_i = (x_i, y_i)$ definiert sind, den Schnittpunkt der Liniensegmente?

Antwort: Es gilt $l = p_1 + \alpha \cdot (p_2 - p_1)$ und $g = p_3 + \beta \cdot (p_4 - p_3)$, sodass wir für den Schnittpunkt $l = g$ erhalten:

$$\begin{aligned} p_1.x + \alpha \cdot (p_2.x - p_1.x) &= p_3.x + \beta \cdot (p_4.x - p_3.x) \quad (*) \\ p_1.y + \alpha \cdot (p_2.y - p_1.y) &= p_3.y + \beta \cdot (p_4.y - p_3.y) \quad (**) \end{aligned}$$

Durch Umformen von $(**)$ ergibt sich:

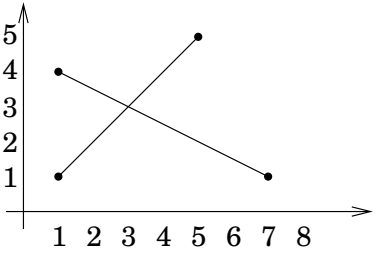
$$\alpha = \frac{(p_3.y - p_1.y) + \beta \cdot (p_4.y - p_3.y)}{p_2.y - p_1.y}$$

Durch Einsetzen in $(*)$ ergibt sich:

$$\beta = \frac{(p_1.x - p_3.x)(p_2.y - p_1.y) + (p_3.y - p_1.y)(p_2.x - p_1.x)}{(p_4.x - p_3.x)(p_2.y - p_1.y) - (p_4.y - p_3.y)(p_2.x - p_1.x)}$$

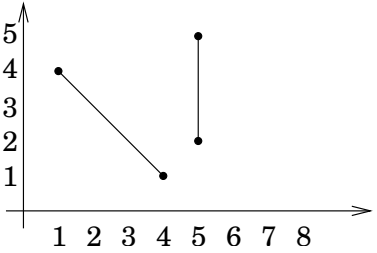
Beispiele zum Segmentschnitt:

$p_1 = (1, 1)$
$p_2 = (5, 5)$
$p_3 = (1, 4)$
$p_4 = (7, 1)$



$\Rightarrow \beta = \frac{1}{3}$

$p_1 = (5, 2)$
$p_2 = (5, 5)$
$p_3 = (1, 4)$
$p_4 = (4, 1)$



$\Rightarrow \beta = \frac{4}{3} > 1$

Dichtestes Punktepaar in der Ebene

gegeben: n Punkte in der Ebene $x_1, x_2, \ldots, x_n \in \mathbb{R}^2$.

gesucht: Ein Punktepaar $x_i, x_j, i \neq j$, dass von allen Paaren den kleinsten Abstand $d(x_i, x_j) = |x_i x_j|$ hat.

Einfacher Algorithmus: Bestimme unter allen $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Paaren das dichteste Punktepaar.

$p_1 := x_1$
$p_2 := x_2$
$min := d(x_1, x_2)$

for $i := 1$ to $n - 1$ do
  for $j := i + 1$ to n do
    if $d(x_i, x_j) < min$
      $min := d(x_i, x_j)$
      $p_1 := x_i$
      $p_2 := x_j$

$\rightarrow$ Laufzeit: $\Theta(n^2)$

Geht es schneller?

Dichtestes Paar einer Zahlenfolge

gegeben: n reelle Zahlen $x_1, x_2, \ldots, x_n \in \mathbb{R}$.

gesucht: Ein Zahlenpaar $x_i, x_j, i \neq j$, dass von allen Paaren den kleinsten Abstand $d(x_i, x_j) = |x_i - x_j|$ hat.

Einfacher Algorithmus: Bestimme unter allen $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Paaren das dichteste Zahlenpaar (wie oben).

$p_1 := x_1$
$p_2 := x_2$
$min := d(x_1, x_2)$

for $i := 1$ to $n - 1$ do
    for $j := i + 1$ to n do
        if $d(x_i, x_j) < min$
            $min := d(x_i, x_j)$
            $p_1 := x_i$
            $p_2 := x_j$

$\rightarrow$ Laufzeit: $\Theta(n^2)$

Geht es schneller?

Dichtestes Paar einer Zahlenfolge

Scan-Line-Verfahren:

$sort(x_1, \ldots, x_n)$
$p_1 := x_1$
$p_2 := x_2$
$min := d(x_1, x_2)$

for $i := 2$ to $n - 1$ do
    if $d(x_i, x_{i+1}) < min$
        $min := d(x_i, x_{i+1})$
        $p_1 := x_i$
        $p_2 := x_{i+1}$

$\rightarrow$ Laufzeit: $\Theta(n \cdot \log(n))$

Kann man diese Idee auch im zweidimensionalen Raum nutzen?

Übersicht

Scan-Line-Prinzip

- Lasse eine vertikale Linie, eine sogenannte Scan-Line, von links nach rechts über eine gegebene Menge von Objekten in der Ebene laufen.
  - $\rightarrow$ Damit zerlegt man ein zweidimensionales geometrisches Problem in eine Folge eindimensionaler Probleme.
- Während man die Scan-Line über die Eingabemenge schwenkt, hält man eine Vertikalstruktur L aufrecht, in der man sich alle für das jeweils zu lösende Problem benötigte Daten merkt.
- Die Scan-Line teilt zu jedem Zeitpunkt die gegebene Menge von Objekten in drei disjunkte Teilmengen:
  - die *toten Objekte*, die bereits vollständig von der Scan-Line überstrichen wurden
  - die gerade *aktiven Objekte*, die gegenwärtig von der Scan-Line geschnitten werden
  - die noch *inaktiven Objekte*, die erst künftig von der Scan-Line geschnitten werden

Übersicht

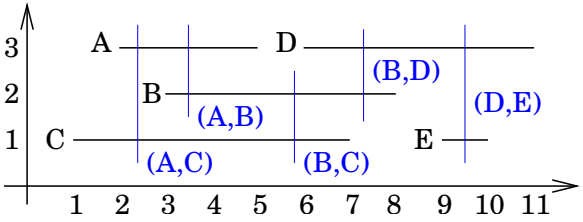
Scan-Line-Prinzip:

- *Sichtbarkeitsproblem*
- Schnittproblem für iso-orientierte Liniensegmente
- Allgemeines Liniensegment-Schnittproblem
- Dichtestes Punktepaar in der Ebene

Motivation: Bei der Kompaktierung höchst-integrierter Schaltkreise müssen Abstandsbedingungen zwischen den relevanten Paaren von Schaltelementen beachtet werden.

- Die Schaltelemente werden abstrahiert durch horizontale Liniensegmente.
- Die relevanten Paare müssen zunächst bestimmt werden.
  → Menge aller Paare, die sich gegenseitig sehen können.

Beispiel:



Sichtbarkeitsproblem

Definition: Zwei Liniensegmente s und s' sind *gegenseitig sichtbar*, wenn es eine vertikale Gerade gibt, die s und s', aber kein anderes Liniensegment zwischen s und s' schneidet.

vorläufig: Alle x-Koordinaten der Anfangs-/Endpunkte sind paarweise verschieden. Später werden wir sehen, wie wir uns von diesen Annahmen frei machen.

Einfacher Algorithmus: Stelle für alle $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Paare von Liniensegmenten fest, ob sie gegenseitig sichtbar sind.

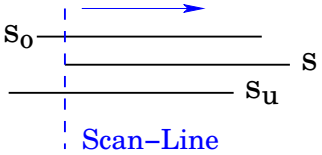
$\rightarrow$ Laufzeit $\Theta(n^2)$, falls der Test auf gegenseitige Sichtbarkeit zweier Liniensegmente in konstanter Zeit erfolgt.

- *Frage:* Wie testet man effizient, ob zwei Liniensegmente gegenseitig sichtbar sind?

Scan-Line-Algorithmus:

$Q :=$ Anfangs-/Endpunkte in aufsteigender x-Reihenfolge
$L := \emptyset$
while $Q \neq \emptyset$ do

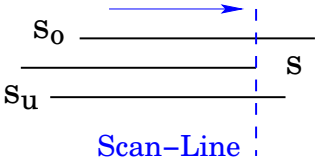
- $p :=$ nächster Punkt aus Q
- falls p ist linker Endpunkt von Segment s



  - füge s in L ein
  - bestimme oberen Nachbarn s_o von s in L
  - bestimme unteren Nachbarn s_u von s in L
  - gib (s, s_o) und (s, s_u) als gegenseitig sichtbar aus

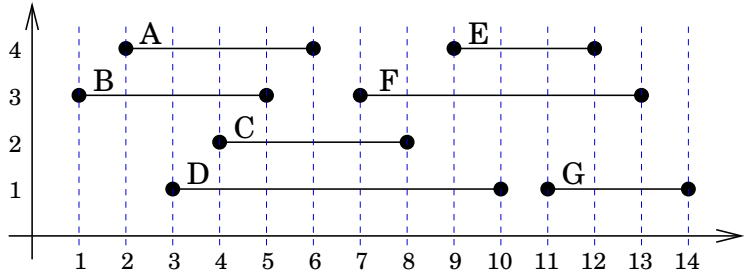
Scan-Line-Algorithmus: (Fortsetzung)

∗ falls p ist rechter Endpunkt von Segment s



· bestimme oberen Nachbarn s_o von s in L
· bestimme unteren Nachbarn s_u von s in L
· entferne s aus L
· gib (s_o, s_u) als gegenseitig sichtbar aus

Übung 59. *Gegeben seien folgende horizontale Liniensegmente:*



Bestimmen Sie mittels des Scan-Line-Verfahrens die gegenseitig sichtbaren Paare von Liniensegmenten.

- *Geben Sie an jedem Haltepunkt der Scan-Line die Datenstruktur L und die dortige Ausgabe an.*
- *Kann es vorkommen, dass Paare doppelt ausgegeben werden?*

Implementierung:

- Die Vertikalstruktur L wird als balancierter Suchbaum implementiert, z.B. als Rot/Schwarz- oder AVL-Baum.
- Die Operationen Einfügen, Entfernen und Bestimmen von Nachbarn werden gut unterstützt. $\rightarrow$ logarithmische Zeit
- Frage: Wie viele Paare gegenseitig sichtbarer Segmente werden höchstens ausgegeben?

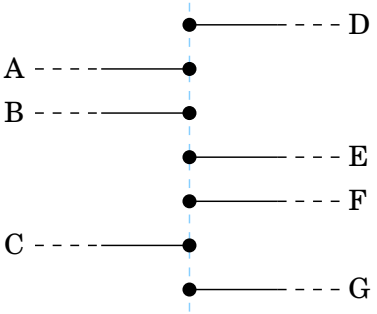
Bemerkung: Es gibt nur $3n - 6$ gegenseitig sichtbare Segmente.

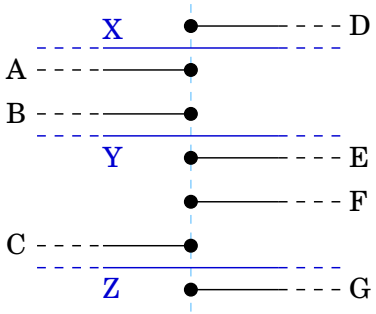
- Stelle die Relation „ist gegenseitig sichtbar“ als planaren Graphen $G = (V, E)$ dar.
  - Jeder Knoten aus V repräsentiert ein Liniensegment,
  - jede Kante aus E stellt ein gegenseitig sichtbares Paar dar.
- Für einen planaren Graphen $G = (V, E)$ gilt: $\mathcal{E} \leq 3 \cdot \mathcal{V} - 6$

$\rightarrow$ Laufzeit $\mathcal{O}(n \cdot \log(n))$, Platz $\mathcal{O}(n)$.

Übung 60. *Welche Änderungen am Algorithmus sind nötig, wenn wir nicht mehr voraussetzen, dass alle Anfangs- und Endpunkte verschiedene x-Koordinaten haben müssen?*

Es wären also folgende Situationen möglich:





Ändert sich durch die Änderungen die asymptotische Laufzeit?

Übersicht

Scan-Line-Prinzip:

- Sichtbarkeitsproblem
- *Schnittproblem für iso-orientierte Liniensegmente*
- Allgemeines Liniensegment-Schnittproblem
- Dichtestes Punktepaar in der Ebene

Schnitt iso-orientierter Liniensegmente

Schnittproblem für iso-orientierte Liniensegmente:

gegeben: Eine Menge von n vertikalen und horizontalen Liniensegmenten in der Ebene.

gesucht: Alle Paare sich schneidender Segmente.

vorläufig: Alle x-Koordinaten der Anfangs-/Endpunkte sowie der vertikalen Segmente sind paarweise verschieden.

Idee: Wenn man sich in der Vertikalstruktur L stets die gerade aktiven horizontalen Segmente merkt, und man dann auf ein vertikales Segment s trifft, dann kann s nur mit den gerade aktiven Elementen Schnittpunkte haben.

Schnitt iso-orientierter Liniensegmente

Scan-Line-Algorithmus:

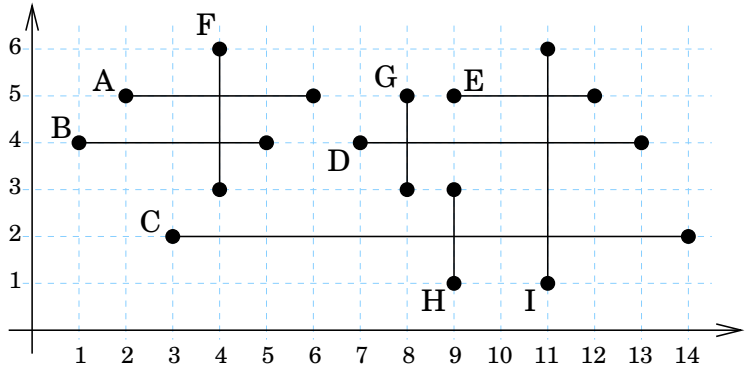
$Q :=$ Anfangs-/Endpunkte horizontaler Segmente und vertikale Segmente in aufsteigender x-Reihenfolge

$L := \emptyset$

while $Q \neq \emptyset$ do

- $p :=$ nächster Punkt aus Q
- falls p ist linker Endpunkt des horizontalen Segments s
  - füge s in L ein
- falls p ist rechter Endpunkt des horizontalen Segments s
  - entferne s aus L
- falls p ist x-Wert eines vertikalen Segments s mit unterem Endpunkt y_u und oberem Endpunkt y_o
  - bestimme alle horizontalen Segmente $t \in L$ mit $y_u \leq y(t) \leq y_o$ und gib (s, t) aus

Übung 61. *Gegeben seien folgende iso-orientierte Liniensegmente:*



Bestimmen Sie mittels des Scan-Line-Verfahrens die Schnittpunkte der Liniensegmente.

Geben Sie an jedem Haltepunkt der Scan-Line die Datenstruktur L und die dortige Ausgabe an.

Implementierung:

- Die Vertikalstruktur L wird als balancierter Blattsuchbaum implementiert, z.B. als Rot/Schwarz- oder AVL-Baum, dessen Blätter verkettet sind.
- Einfügen und Entfernen ist in Zeit $\mathcal{O}(\log(n))$ möglich.
- Bereichsanfragen können in Zeit $\mathcal{O}(\log(n) + r)$ beantwortet werden, wobei r die Anzahl der Elemente in dem angefragten Bereich ist.

$\rightarrow$ Laufzeit: $\mathcal{O}(n \cdot \log(n) + k)$, wobei k die Anzahl der Schnittpunkte insgesamt ist.

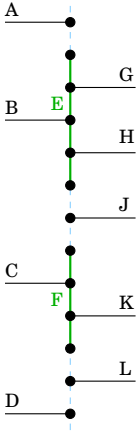
Platzbedarf: $\mathcal{O}(n)$

Übung 62.

Welche Änderungen am Algorithmus sind nötig, wenn wir nicht mehr voraussetzen, dass alle Anfangs- und Endpunkte sowie die vertikalen Segmente verschiedene x-Koordinaten haben müssen?

Es wäre also eine Situation wie rechts abgebildet möglich.

Ändert sich durch die Änderungen die asymptotische Laufzeit?



Übersicht

Scan-Line-Prinzip:

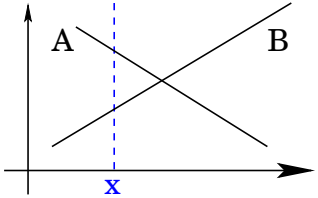
- Sichtbarkeitsproblem
- Schnittproblem für iso-orientierte Liniensegmente
- *Allgemeines Liniensegment-Schnittproblem*
- Dichtestes Punktepaar in der Ebene

Allgemeines Liniensegment-Schnittproblem

gegeben: Eine Menge von n Liniensegmenten.

gesucht: Alle Paare sich schneidender Segmente.

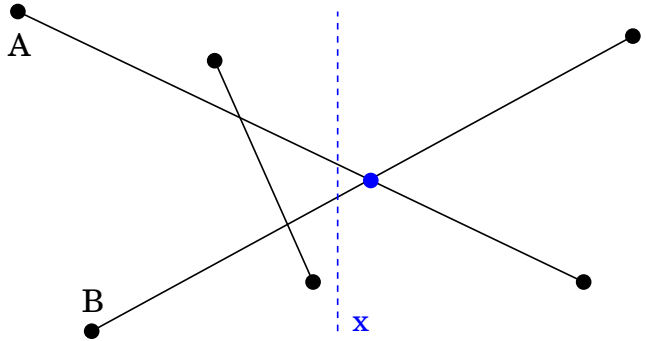
Vereinbarung: *A liegt x-oberhalb von B*, wenn die vertikale Gerade durch x sowohl A als auch B schneidet und der Schnittpunkt von x mit A oberhalb des Schnittpunktes von x mit B liegt. Schreibweise: $A \uparrow_x B$



vorläufig:

- Es gibt keine vertikalen Liniensegmente.
- In jedem Punkt schneiden sich höchstens zwei Segmente.
- Alle Anfangs-/Endpunkte haben paarweise verschiedene x-Koordinaten.

Beobachtung: Wenn sich zwei Liniensegmente A und B schneiden, dann gibt es eine Stelle x links vom Schnittpunkt, sodass A und B in der Ordnung $\uparrow_x$ unmittelbar aufeinanderfolgen.



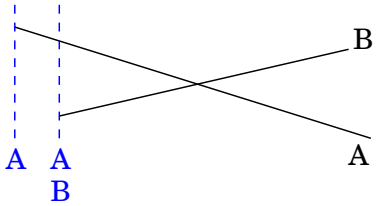
Hier geht die Voraussetzung ein, dass sich in jedem Punkt höchstens zwei Segmente schneiden.

Allgemeines Liniensegment-Schnittproblem

Scan-Line-Algorithmus:

$Q :=$ Anfangs-/Endpunkte in aufsteigender x-Reihenfolge
$L := \emptyset$
while $Q \neq \emptyset$ do

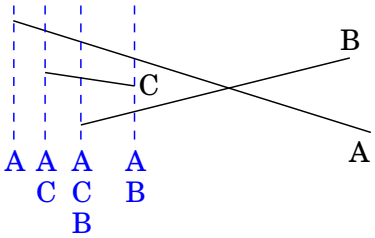
- $p :=$ nächster Punkt aus Q
- falls p ist linker Endpunkt von Segment s



  - füge s entsprechend $\uparrow_{p.x}$ in L ein
  - bestimme oberen Nachbarn s_o von s in L
  - bestimme unteren Nachbarn s_u von s in L
  - füge ggf. $s \cap s_o$ bzw. $s \cap s_u$ in Q ein

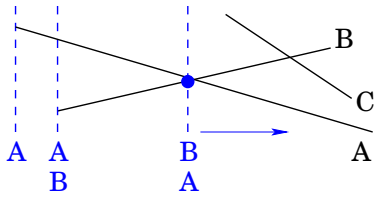
Scan-Line-Algorithmus: (Fortsetzung)

∗ falls p ist rechter Endpunkt von Segment s



· bestimme oberen Nachbarn s_o von s in L
· bestimme unteren Nachbarn s_u von s in L
· entferne s aus L
· füge ggf. Schnittpunkt $s_o \cap s_u$ in Q ein

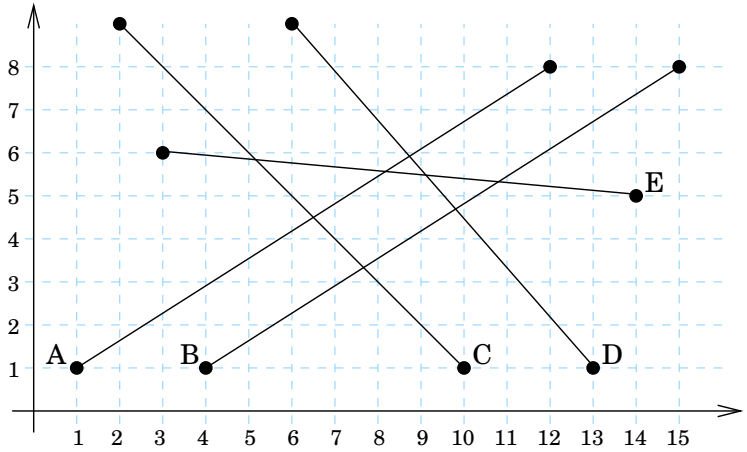
Wenn die Scan-Line einen Schnittpunkt zweier Segmente A und B passiert, dann wechseln A und B ihren Platz in L, damit die lokale Ordnung der Vertikalstruktur L korrekt bleibt.



Scan-Line-Algorithmus: (Fortsetzung)

- falls p Schnittpunkt von s und t ist mit $s \uparrow_{p.x} t$
  - gib (s, t) mit Schnittpunkt p aus
  - vertausche s und t in L
  - bestimme oberen Nachbarn t_o von t in L
  - füge ggf. Schnittpunkt $t \cap t_o$ in Q ein
  - bestimme unteren Nachbarn s_u von s in L
  - füge ggf. Schnittpunkt $s \cap s_u$ in Q ein

Übung 63. *Gegeben seien folgende Liniensegmente:*



- *Bestimmen Sie mittels des Scan-Line-Verfahrens alle Schnittpunkte der Segmente.*
- *Geben Sie an den Haltepunkten der Scan-Line die Struktur L und die Ausgabe an.*

Implementierung:

- Q wird als balancierter Suchbaum implementiert:
  - Füge Schnittpunkt S nur ein, falls noch nicht vorhanden.
  - Suchen, Einfügen, Entfernen und Bestimmen des kleinsten Wertes werden gut unterstützt. $\rightarrow$ logarithmische Laufzeit
- L wird ebenfalls als balancierter Suchbaum implementiert:
  - Einfügen, Entfernen und Bestimmen von direkten Nachbarn werden gut unterstützt. $\rightarrow$ logarithmische Laufzeit
- *Platz:* $\mathcal{O}(n+k)$, wobei k die Anzahl der Schnittpunkte ist.
- Die Schleife wird für k Schnittpunkte höchstens $(2n+k)$-mal durchlaufen und wir erhalten als *Laufzeit:* $\mathcal{O}((n+k)\cdot\log(n))$

Frage: Wenn der Platzbedarf $\mathcal{O}(n+k)$ ist, müsste die Laufzeit dann nicht $\mathcal{O}((n+k)\cdot\log(n+k))$ sein?

Bemerkung:

- Die Komplexitäten sind nur akzeptabel für kleine Wert von k.
- Es gibt Verfahren[33] mit Laufzeit $\mathcal{O}(n \cdot \log(n) + k)$.

Platzersparnis: Speichere nicht alle Schnittpunkte.

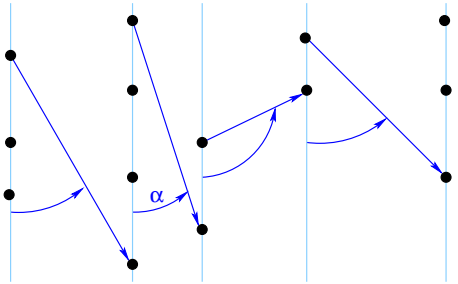
- Wir nehmen für jedes aktive Liniensegment s nur den am weitesten links liegenden Schnittpunkt in Q auf.
- Wird ein weiter links liegender Schnittpunkt mit Segment s gefunden, so wird dieser eingefügt und der alte entfernt.
- Um für jedes aktive Segment s leicht feststellen zu können, ob schon ein Schnittpunkt in Q enthalten ist, an dem s beteiligt ist, vermerke zu s einen Zeiger auf den Schnittpunkt.

$\rightarrow$ *Platz:* $\mathcal{O}(n)$

[33] Chazelle, Edelsbrunner: An optimal algorithm for intersecting line segments in the plane. Journal of the ACM, 1992. https://dl.acm.org/doi/10.1145/147508.147511

Um die vereinfachenden Annahmen fallen lassen zu können, sind bspw. folgende Änderungen am Algorithmus nötig:

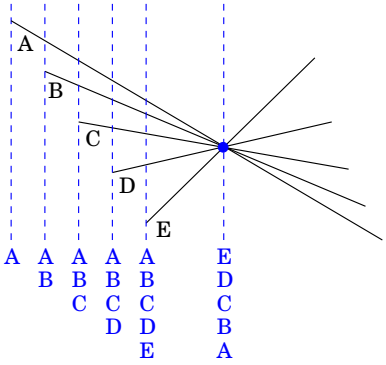
1. Anfangs- und Endpunkte haben gleiche x-Koordinaten.
   - Transformation des Koordinatensystems: Einteilung der Punkte in Gruppen mit gleicher x-Koordinate.



   - Jeweils vom oberen Punkt einer Gruppe zum unteren Punkt der benachbarten rechten Gruppe eine Gerade legen. Sei α der kleinste Winkel einer solchen Geraden mit der y-Achse.
   - Drehe Koordinatensystem um $\alpha/2$ entgegen den Uhrzeigersinn.

2. Schnittpunkte können aber noch mit Anfangs- oder Endpunkten zusammen fallen:
   - Bearbeite zunächst die linken Endpunkte, dann die Schnittpunkte und danach die rechten Endpunkte.

3. In einem Punkt schneiden sich mehr als zwei Segmente.
   - Verfahre wie bisher und teste jedes neu entdeckte Segment auf Schnitt mit seinen beiden Nachbarn.
   - Ordne die Schnittereignisse in Q lexikographisch an.
   - Berichte den mehrfachen Schnitt und invertiere die Reihenfolge der beteiligten Segmente en bloc.



Ändert sich durch diese Änderungen die asymptotische Laufzeit?

Übersicht

Scan-Line-Prinzip:

- Sichtbarkeitsproblem
- Schnittproblem für iso-orientierte Liniensegmente
- Allgemeines Liniensegment-Schnittproblem
- *Dichtestes Punktepaar in der Ebene*

Dichtestes Punktepaar in der Ebene

Kommen wir zurück zu unserem anfänglichen Problem.

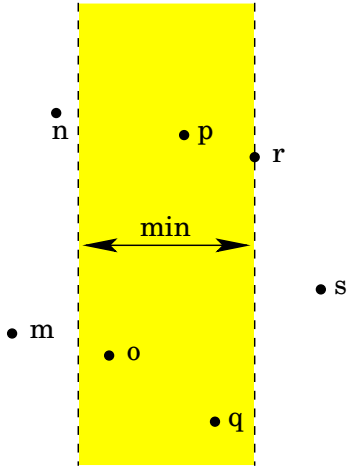
Wenn r mit einem Punkt p links von r ein Paar bilden soll, dann muss p im Innern des senkrechten Streifens der Breite *min* liegen.

Wir merken uns in L die Punkte innerhalb des Streifens.

Die Datenstruktur L muss aktualisiert werden, wenn

- der linke Streifenrand über einen Punkt hinweg wandert.

$\rightarrow$ Der betroffene Punkt wird aus L entfernt.



- der rechte Streifenrand (Scan-Line) auf einen neuen Punkt stößt.

→ Der neue Punkt muss in L aufgenommen werden und es ist zu testen, ob der neue Punkt mit einem der Punkte innerhalb des Streifens einen Abstand kleiner als *min* hat.

falls ja: *min* und die Streifenbreite müssen auf den neuen Wert verringert werden. Das wiederum kann eine Folge von Ereignissen des ersten Typs bedeuten.

oben: Der Streifen schrumpft auf die Breite $d(p, r)$, wenn die Scan-Line Punkt r erreicht. Der linke Streifenrand springt dann über Punkt o hinweg.

Punkte betreten und verlassen den Streifen in der Reihenfolge von links nach rechts:

- Wir sortieren daher die Punkte nach aufsteigenden x-Koordinaten und speichern sie in einem Array.
- Wir testen dabei, ob bei Punkten mit gleichem x-Wert auch die y-Koordinate übereinstimmt. (falls ja $\rightarrow$ terminiere)

Wir arbeiten mit zwei Positionen l und r:

- $Q[l]$: der am weitesten links liegende Punkt im senkrechten Streifen
- $Q[r]$: nächster Punkt, auf den die Scan-Line treffen wird

$\rightarrow$ $Q[l]$ verlässt den Streifen, wenn $Q[l].x + min \leq Q[r].x$ gilt.

Dichtestes Punktepaar in der Ebene

Scan-Line-Algorithmus:

$Q :=$ Folge der n Punkte in aufsteigender x-Reihenfolge
$L := \{p_1, p_2\}$
$min := d(p_1, p_2)$
$l := 1$
$r := 3$

while $r \leq n$ do
- if $Q[l].x + min \leq Q[r].x$ then
  - entferne $Q[l]$ aus L
  - $l := l + 1$
- else
  - $min := MinDist(L, Q[r], min)$
  - füge $Q[r]$ in L ein
  - $r := r + 1$

Dichtestes Punktepaar in der Ebene

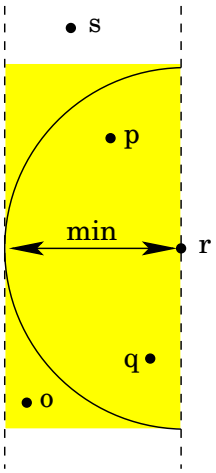
MinDist(L, r, min) liefert die minimale Distanz vom neuen Punkt r auf der Scan-Line zu allen übrigen Punkten im senkrechten Streifen L, oder ggf. den bisherigen Wert *min*.

Inspiziere nur diejenigen Punkte im Streifen, deren y-Koordinate um höchstens *min* oberhalb oder unterhalb von $r.y$ liegen.

L muss Einfügen und Entfernen von Punkten sowie Bereichsanfragen unterstützen:

→ nach y-Koordinaten geordneter, balancierter Suchbaum mit doppelt verketteten Blättern

Platz: $\mathcal{O}(n)$ (es werden nur Punkte gespeichert)
Zeit: ???



Zeit:

$$\mathcal{O}(n \cdot \log(n) + \sum_{i=1}^{n} k_i)$$

denn:

- Die while-Schleife wird n-mal durchlaufen.
- Einfügen in bzw. entfernen aus L erfolgt in Zeit $\mathcal{O}(\log(n))$.
- Der i-te Aufruf von *MinDist* kostet Zeit $\mathcal{O}(\log(n) + k_i)$, dabei bezeichnet k_i die Größe der Antwort der i-ten Bereichsanfrage.

Abschätzung:

- naiv: Für $k_i \leq i$ ergibt sich als Laufzeit $\mathcal{O}(n^2)$.
- bessere Abschätzung ???

Dichtestes Punktepaar in der Ebene

Idee für bessere Abschätzung:

- Alle Punkte links der Scan-Line, insbesondere die im Rechteck, haben einen Abstand von mindestens *min* voneinander.

$\rightarrow$ Es passen nicht allzuviele Punkte ins Rechteck hinein!

Lemma: Sei $M > 0$ und P eine Menge von Punkten in der Ebene, von denen je zwei mindestens den Abstand M voneinander haben. Dann enthält ein Rechteck mit Kantenlängen M und $2M$ höchstens 10 Punkte aus P.

Damit gilt $k_i \leq 10$ für alle i und wir erhalten als Laufzeit des Verfahrens: $\mathcal{O}(n \cdot \log(n))$

Dichtestes Punktepaar in der Ebene

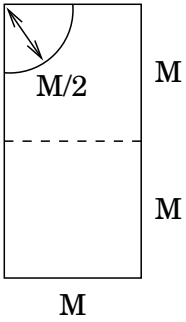
Beweis: Nach Voraussetzung gilt: Die Kreisumgebungen $U_{M/2}(p)$ der Punkte $p \in P$ sind paarweise disjunkt.

Liegt der Punkt p im Rechteck R, so ist mindestens $\frac{1}{4}$ der offenen Kreisscheibe in R enthalten.

Durch Flächenberechnung ergibt sich, dass das Rechteck R höchstens

$$\frac{\text{Fläche}(R)}{\text{Fläche Viertelkreis}} = \frac{2M^2}{\frac{1}{4}\pi(\frac{M}{2})^2} = \frac{32}{\pi} < 11$$

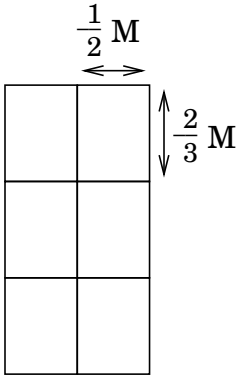
viele Punkte aus P enthalten kann!



Dichtestes Punktepaar in der Ebene

Bemerkung: Obiges Lemma ist auch für 6 Punkte korrekt!

Teile das Rechteck R in sechs gleich große Rechtecke mit Kantenlänge $\frac{2}{3}M$ und $\frac{1}{2}M$ ein.



Zwei Punkte in einem der Rechtecke können höchstens so weit voneinander entfernt sein, wie seine Diagonale lang ist, also höchstens

$$\sqrt{\left(\frac{2}{3}M\right)^2 + \left(\frac{1}{2}M\right)^2} \;=\; \frac{5}{6}M \;<\; M.$$

Also kann in jedem Rechteck höchstens ein Kreismittelpunkt sein.

Übersicht

Konvexe Hülle

Definition:

- Eine Menge $S \subseteq \mathbb{R}^2$ heißt konvex, wenn für je zwei Punkte $p, q \in S$ auch $\overline{pq} \subseteq S$ gilt.
- Die konvexe Hülle $CH(S)$ von S ist die kleinste konvexe Menge, die S enthält.

Anschaulich:

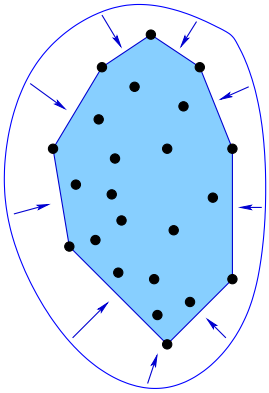
- Lege ein großes Gummiband um alle Punkte und lass es los!

$\rightarrow$ hilft algorithmisch leider gar nicht

In der Mathematik:

- Definiere $CH(S) := \bigcap_{C \supseteq S : C \text{ konvex}} C$.

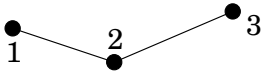
$\rightarrow$ hilft leider auch nicht



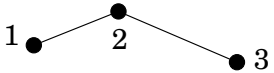
Konvexe Hülle

Motivation: Die konvexe Hülle hat deutlich weniger Punkte als das Original, wodurch sich Rechenzeit sparen lässt.

- Kollisionstest bei Computerspielen:
  - Zunächst mit den konvexen Hüllen der Objekte auf Kollision testen.
  - Wenn zwei Hüllen sich nicht berühren, dann berühren sich auch die Ausgangsformen nicht.
- Sichtbare Flächen beim Rendern:
  - Zunächst die Umgebung darstellen, danach die konvexen Hüllen der Figuren.
  - Wenn die Hülle einer Person nicht sichtbar ist, rendern wir die komplexe Figur nicht.



Linksdrehung



Rechtsdrehung

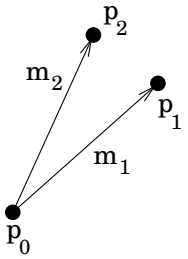
Algorithmus:

- Suche einen Punkt p mit kleinster y-Koordinate. Falls es mehrere gibt, wähle den mit kleinster x-Koordinate.
- Sortiere alle Punkte aufsteigend nach Winkeln zu einer gedachten Horizontalen mit Anfang p.
- $v := p$
- solange $next(v) \neq p$ tue
  falls $[v, next(v), next(next(v))]$ ist Linksdrehung dann
  - $v := next(v)$

  sonst
  - lösche $next(v)$
  - $v := pred(v)$

gegeben: Drei Punkte $p_0, p_1, p_2 \in \mathbb{R}^2$.

Frage: Ist der Weg p_0, p_1, p_2 eine Linksdrehung?



$$m_1 = \frac{dy_1}{dx_1} \qquad m_2 = \frac{dy_2}{dx_2}$$

$$\begin{aligned} dx_1 &= p_1.x - p_0.x & \qquad dx_2 &= p_2.x - p_0.x \\ dy_1 &= p_1.y - p_0.y & \qquad dy_2 &= p_2.y - p_0.y \end{aligned}$$

$\longrightarrow$ Linksdrehung, wenn $m_1 < m_2$

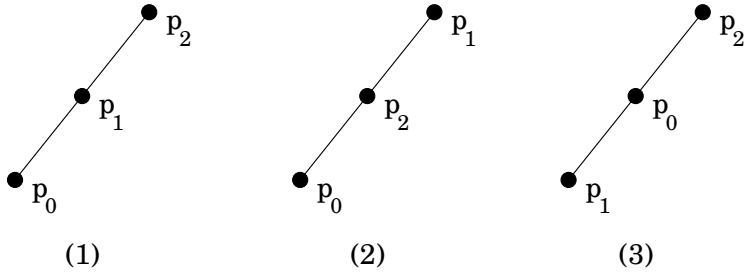
Problem, falls $dx_1 = 0$ oder $dx_2 = 0$ ist!
Multipliziere beide Seiten mit $dx_1 \cdot dx_2$:

$$m_1 < m_2 \iff dy_1 \cdot dx_2 < dy_2 \cdot dx_1$$

oder anders gesagt:

$$\begin{vmatrix} p_0.x & p_0.y & 1 \\ p_1.x & p_1.y & 1 \\ p_2.x & p_2.y & 1 \end{vmatrix} > 0$$

Was ist, wenn die drei Punkte auf einer Geraden liegen, also $m_1 = m_2$?

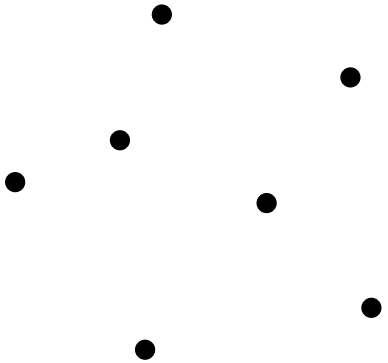


(1) o.k. für Graham-Scan

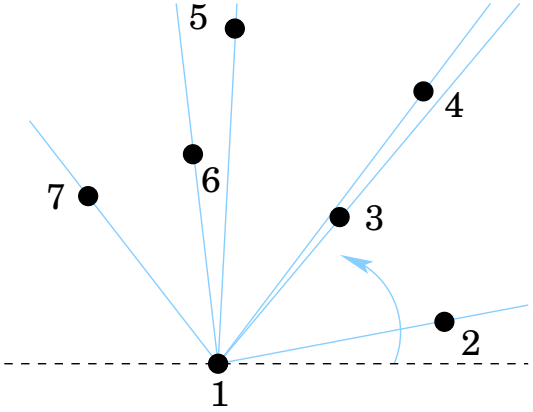
(2) $|p_0p_1| \geq |p_0p_2| \iff dx_1^2 + dy_1^2 \geq dx_2^2 + dy_2^2$

(3) $dx_1 \cdot dx_2 < 0$ oder $dy_1 \cdot dy_2 < 0$

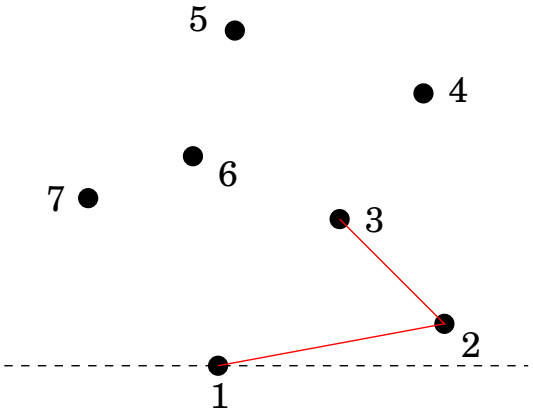
Graham Scan



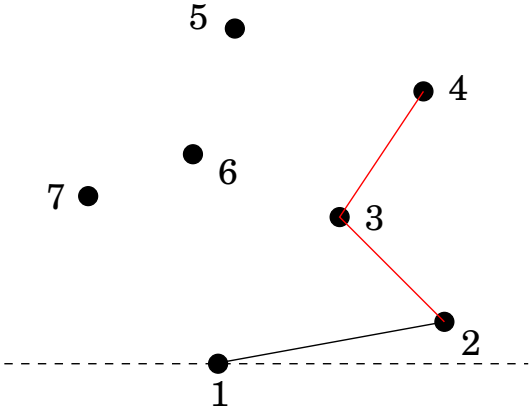
Graham Scan



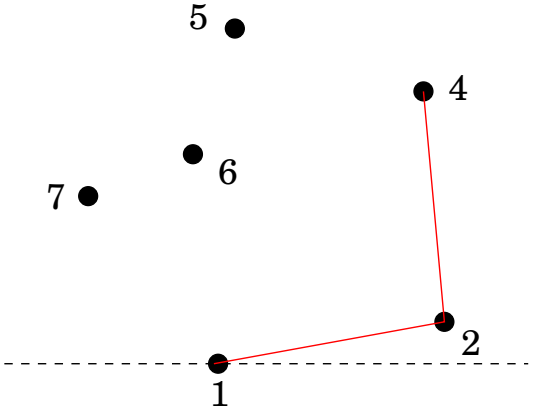
Graham Scan



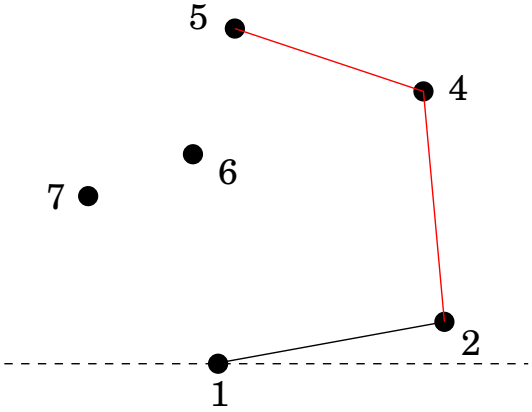
Graham Scan



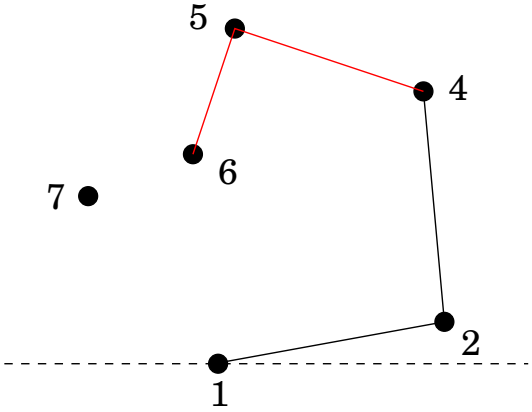
Graham Scan



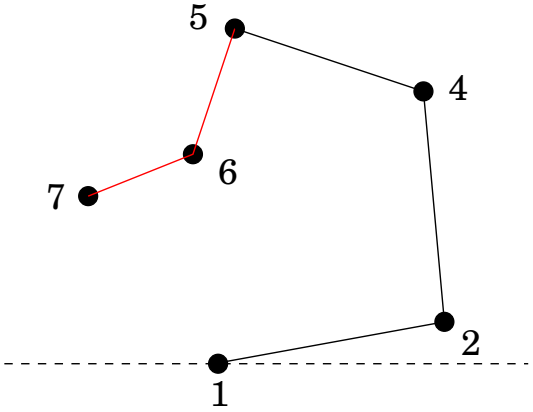
Graham Scan



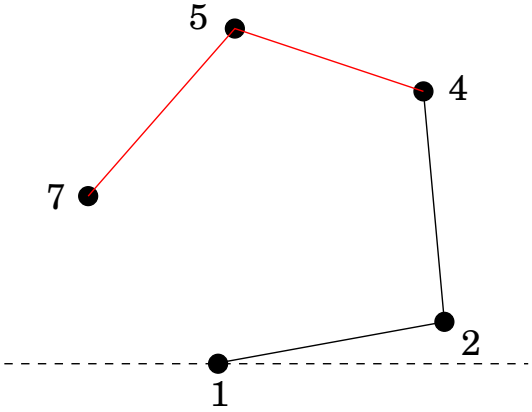
Graham Scan



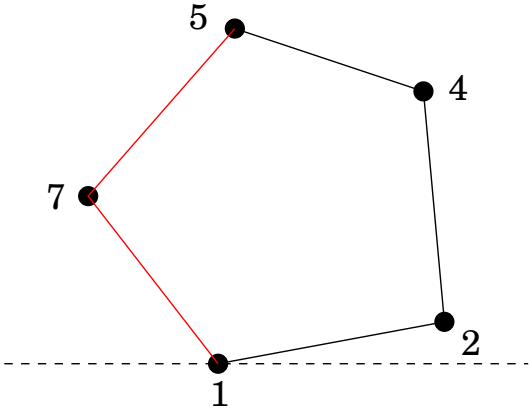
Graham Scan



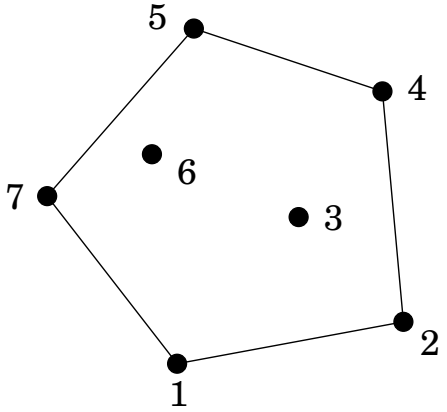
Graham Scan



Graham Scan



Graham Scan



Graham Scan

Implementierung:

- Die Punkte werden in einer doppelt verketteten Liste mit *next*- und *pred*-Zeigern gespeichert.

Komplexität:

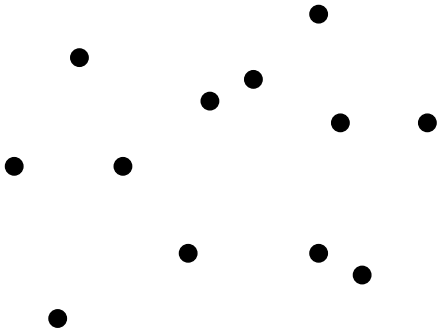
- Laufzeit: $\mathcal{O}(n \cdot \log(n))$
- Platz: $\mathcal{O}(n)$

Package Wrapping

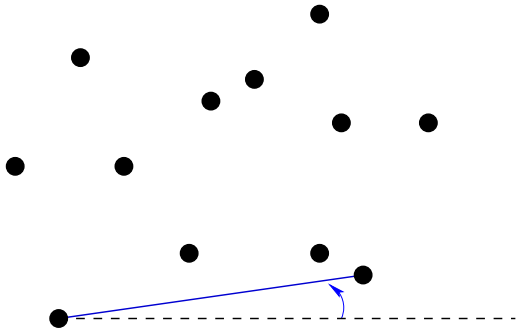
auch Jarvis' March genannt:

* Suche einen Punkt p mit kleinster y-Koordinate. Falls mehrere existieren: Wähle den mit kleinster x-Koordinate.
* $p' := p$
* solange (p' ist kein Punkt mit größter y-Koordinate) tue
  - Suche p'', sodass $\overline{p'p''}$ den kleinsten Winkel zu $\overline{p'q}$ hat, wobei $y(q) = y(p')$ und $x(q) > x(p')$ gilt.
  - füge p'' in $CH(S)$ ein
  - $p' := p''$
* solange ($p' \neq p$) tue
  - Suche p'', sodass $\overline{p'p''}$ den kleinsten Winkel zu $\overline{p'q}$ hat, wobei $y(q) = y(p')$ und $x(q) < x(p')$ gilt.
  - füge p'' in $CH(S)$ ein
  - $p' := p''$

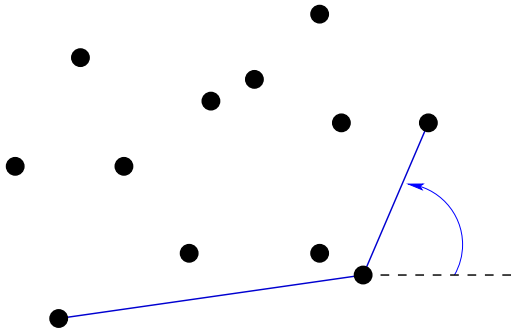
Package Wrapping



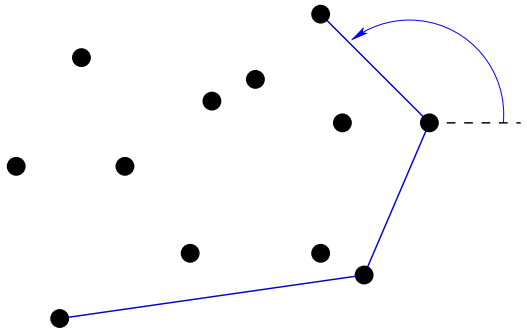
Package Wrapping



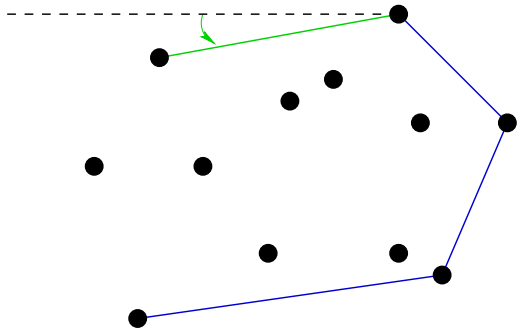
Package Wrapping



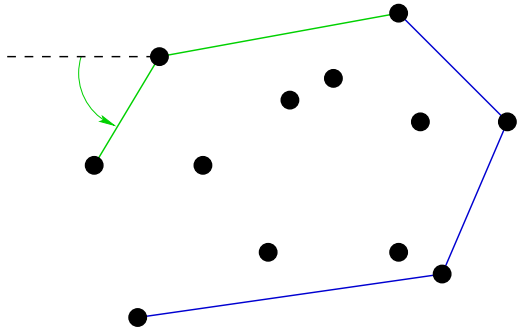
Package Wrapping



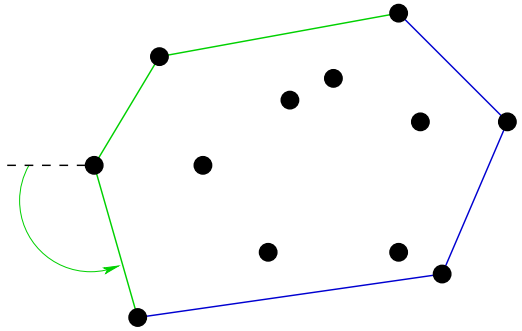
Package Wrapping



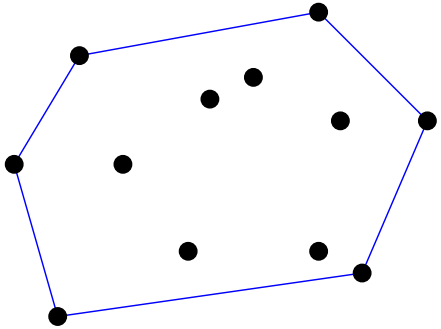
Package Wrapping



Package Wrapping



Package Wrapping



Implementierung:

- Durch die getrennte Berechnung des linken und rechten Teils der konvexen Hülle ist es möglich, die kleinsten Winkel zu berechnen, ohne trigonometrische Funktionen zu verwenden. Dabei wird auf die Idee der Linksdrehung zurück gegriffen.

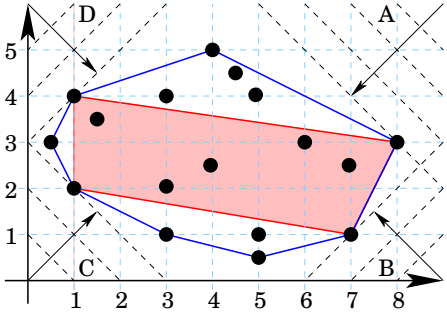
Komplexität:

- Laufzeit: $\mathcal{O}(|CH| \cdot n)$
  $\rightarrow$ Wenn alle Punkte auf der konvexen Hülle liegen: $\mathcal{O}(n^2)$
- Platz: $\mathcal{O}(n)$

Interior Elimination

Package Wrapping bzw. Jarvis' March kann beschleunigt werden, wenn zuerst einige der Punkte, die sicher nicht zur konvexen Hülle gehören, aus der Punktemenge P entfernt werden.

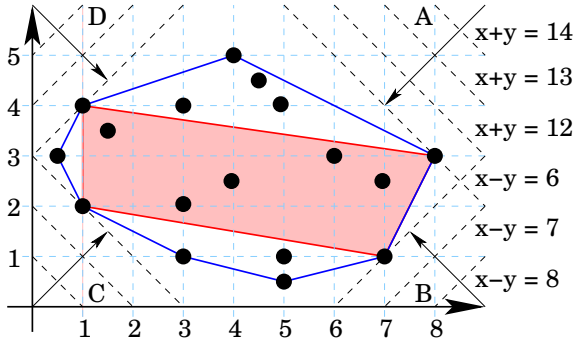
- Bei gleichverteilten x,y-Werten bleiben im Mittel nur $\mathcal{O}(\sqrt{n})$ Punkte erhalten.
- Im worst-case ergibt sich keine Verbesserung.



Fragen:

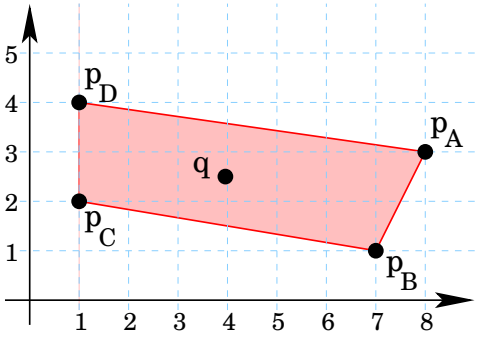
- Wie findet man effizient die 4 Punkte auf der konvexen Hülle?
- Wann liegt ein Punkt innerhalb des Vierecks?
- Wie entfernt man effizient die Punkte innerhalb des Vierecks aus der Punktemenge P?

Wie findet man effizient die 4 Punkte auf der konvexen Hülle?



- für p_A gilt: $p_A.x + p_A.y \geq q.x + q.y \ \forall q \in P$
- für p_B gilt: $p_B.x - p_B.y \geq q.x - q.y \ \forall q \in P$
- für p_C gilt: $p_C.x + p_C.y \leq q.x + q.y \ \forall q \in P$
- für p_D gilt: $p_D.x - p_D.y \leq q.x - q.y \ \forall q \in P$

Wann liegt ein Punkt q innerhalb des Vierecks?



- p_A, p_B, q ist Rechtsdrehung
- p_B, p_C, q ist Rechtsdrehung
- p_C, p_D, q ist Rechtsdrehung
- p_D, p_A, q ist Rechtsdrehung

Interior Elimination

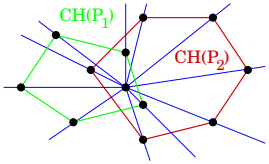
Wie entfernt man effizient die Punkte innerhalb des Vierecks aus der Punktemenge P?

- Wir gehen davon aus, dass die Punkte in einem Array gespeichert sind.
- Algorithmus:

  $i := 1$
  $last := n$
  solange $i < last$ tue
  falls p_i innerhalb des Rechtecks ist, dann
  · swap(p_i, p_{last})
  · $last := last - 1$
  sonst
  · $i := i + 1$

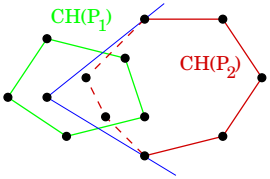
- *Divide:* Teile $P = \{p_1, \ldots, p_n\}$ auf in zwei etwa gleich große Mengen $P_1 = \{p_1, \ldots, p_{\lceil n/2 \rceil}\}$ und $P_2 = \{p_{\lceil n/2 \rceil + 1}, \ldots, p_n\}$.
- *Conquer:* Berechne $CH(P_1)$ und berechne $CH(P_2)$ rekursiv.
- *Merge:* Finde einen Punkt p innerhalb von $CH(P_1)$.
  - falls p innerhalb von $CH(P_2)$ liegt:

    Mische $CH(P_1)$ und $CH(P_2)$ zu einer Sortierung nach Winkeln bzgl. eines Segments mit Endpunkt p.

  - sonst liegt p außerhalb von $CH(P_2)$:

    Bestimme den Keil von p und q_1, q_2 aus $CH(P_2)$, sodass alle Punkte aus P_2 in diesem Keil liegen. Mische $CH(P_1)$ und die äußere Kontur von $CH(P_2)$.

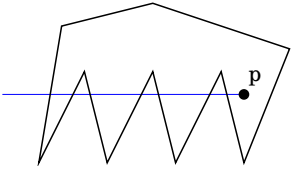
    

  - Wende auf die gemischte Liste Graham Scan an.

Anmerkungen:

- Mischen der beiden konvexen Hüllen in Zeit $\mathcal{O}(|CH(P_1)| + |CH(P_2)|) = \mathcal{O}(n)$.
- Anwendung von Graham Scan in Zeit $\mathcal{O}(n)$, da Sortieren nach Winkeln entfällt.
- Die Entscheidung, ob ein Punkt innerhalb oder außerhalb eines Polygons liegt, kann in Zeit $\mathcal{O}(n)$ getroffen werden:
  - Anzahl Schnittpunkte ungerade: p liegt innerhalb
  - sonst: p liegt außerhalb
  - es sind einige Sonderfälle zu beachten:
    - Linie schneidet Eckpunkt oder
    - eine Kante liegt auf der Linie

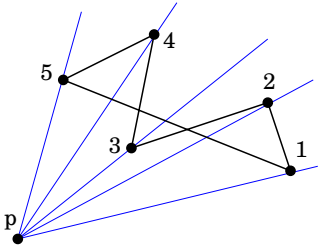
  $\rightarrow$ Punkt-in-Polygon-Test nach Jordan



- Komplexität:
  - Laufzeit: $T(n) = 2 \cdot T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n \cdot \log(n))$
  - Platz: $\mathcal{O}(n)$

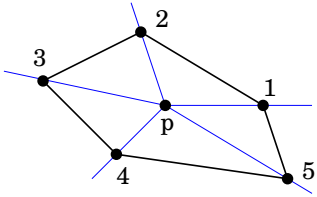
`https://de.wikipedia.org/wiki/Punkt-in-Polygon-Test_nach_Jordan`

Es ist wichtig, dass der Punkt p, mit dessen Hilfe die Sortierung der Winkel erfolgt, innerhalb, oder zumindest auf der zu bestimmenden konvexen Hülle liegt.



p liegt außerhalb der zu berechnenden konvexen Hülle:

Es ergibt sich ein geschlossener Weg, bei dem sich einige Kanten schneiden. Ungeeignet für Graham Scan.



p liegt innerhalb der zu berechnenden konvexen Hülle:

Es ergibt sich ein geschlossener Weg, bei dem sich keine Kanten schneiden. Geeignet für Graham Scan.