

Theoretische Informatik 3

Johannes Köbler



Institut für Informatik
Humboldt-Universität zu Berlin

SS 2011

Termine

Vorlesung



Di	9:15-10:45	wöch.	RUD 25	3.101	W. Kössler
----	------------	-------	--------	-------	------------

Übungen (ab 19.04.2011)



Di	11:15-12:45	14tg./1.	RUD 25	3.101	W. Kössler
----	-------------	----------	--------	-------	------------



Di	11:15-12:45	14tg./2.	RUD 25	3.101	W. Kössler
----	-------------	----------	--------	-------	------------



Do	09:15-10:45	14tg./1.	RUD 25	3.101	W. Kössler
----	-------------	----------	--------	-------	------------



Do	09:15-11:45	14tg./2.	RUD 25	3.101	W. Kössler
----	-------------	----------	--------	-------	------------

• Tutoren: Frank Fuhlbrück und Stephan Verbücheln

(siehe auch www.informatik.hu-berlin.de/forschung/gebiete/algorithmenII/Lehre/ss11/theo3)

Übungen (Anmeldung über GOYA erforderlich)

Ausgabe der Aufgabenblätter

- in der VL, auf GOYA und der VL-Webseite

Bearbeitung

- in Gruppen von maximal **zwei** Teilnehmern
- bitte Namen und Matrikelnr. angeben

Abgabe

- bis **9:10 Uhr** im Erdgeschoss von Haus 4!
(Abgabetermin siehe Aufgabenblatt)
- bitte nur in Papierform

Rückgabe

- in den Übungsgruppen

Scheinkriterien

- Lösen von $\geq 50\%$ der schriftlichen Aufgaben,
- **Erfolgreiches** Vorrechnen von mindestens zwei mündl. Aufgaben.

Klausur

- **Zulassung** zur Klausur nur mit Übungsschein.
- Klausur: Mo, 18.07.2011, 12.00-14:00 Uhr in RUD 26, 0'115.
- Nachklausur: erste Woche des Wintersemesters

Gibt es zum organisatorischen Ablauf noch Fragen?

Zentrale Themen von Th1 3

Themen der VL Theoretische Informatik 1

- Mathematische Grundlagen der Informatik, Beweise führen, Modellierung Aussagenlogik, Prädikatenlogik
- Welche Probleme sind lösbar? Berechenbarkeitstheorie

Themen der VL Theoretische Informatik 2

- Welche Rechenmodelle sind adäquat? Automatentheorie
- Welcher Aufwand ist nötig? Komplexitätstheorie

Themen der VL Theoretische Informatik 3

- Wie lassen sich eine Reihe von praktisch relevanten Problemstellungen möglichst effizient lösen?
- Wie lässt sich die Korrektheit von Algorithmen beweisen und wie lässt sich ihre Laufzeit abschätzen? Algorithmik

Algorithmen

Der Begriff des Algorithmus

- Der Begriff **Algorithmus** geht auf den persischen Gelehrten **Muhammed Al Chwarizmi** (8./9. Jhd.) zurück.
- Ältester bekannter nicht-trivialer Algorithmus:
Euklidischer Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen (300 v. Chr.).
- Von einem Algorithmus wird erwartet, dass er bei jeder zulässigen **Problemeingabe** nach endlich vielen Rechenschritten eine korrekte **Ausgabe** liefert.
- Ein Algorithmus ist ein „Verfahren“ zur Lösung eines Berechnungsproblems, das sich prinzipiell auf einer Turingmaschine implementieren lässt (**Church-Turing-These**).

Maschinenmodell

Die Registermaschine (engl. random access machine; RAM)

- Bei Aussagen zur Laufzeit von Algorithmen beziehen wir uns auf die Registermaschine.
- Dieses Modell ist etwas flexibler als die Turingmaschine, da es den unmittelbaren Lese- und Schreibzugriff (**random access**) auf eine beliebige Speichereinheit (Register) erlaubt.
- Als Speicher stehen beliebig viele Register zur Verfügung, die jeweils eine beliebig große natürliche Zahl speichern können.
- Auf den Registerinhalten sind folgende arithmetische Operationen in einem Rechenschritt ausführbar:
 - Addition,
 - Subtraktion,
 - abgerundetes Halbieren und Verdoppeln.
- Unabhängig davon geben wir die Algorithmen in Pseudocode an. Das RAM-Modell benutzen wir nur zur Komplexitätsabschätzung.

Laufzeit von RAM-Programmen

Laufzeit

- Wie bei TMs wird die Laufzeit in der Länge der Eingabe gemessen.
- **BEACHTEN:** Bei arithmetischen Problemen (Multiplikation, Division, Primzahltests, etc.) wird die Länge einer Zahleingabe n durch die Anzahl $\lceil \log n \rceil$ der für die **Binärkodierung** von n benötigten Bits gemessen.
- Bei nicht-arithmetischen Problemen (z.B. Graphalgorithmen oder Sortierproblemen) bestimmt lediglich die Anzahl der gegebenen Zahlen die Länge der Eingabe.

Asymptotische Laufzeit und Landau-Notation

Zur Erinnerung:

Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{R}^+$.

- Wir schreiben $f(n) = \mathcal{O}(g(n))$, falls es Zahlen n_0 und c gibt mit
$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Bedeutung: „ f wächst **nicht wesentlich schneller** als g .“

- Formal bezeichnet der Term $\mathcal{O}(g(n))$ die Klasse aller Funktionen f , die obige Bedingung erfüllen.
- Die Gleichung $f(n) = \mathcal{O}(g(n))$ drückt also in Wahrheit eine **Element-Beziehung** $f \in \mathcal{O}(g(n))$ aus.
- $\mathcal{O}$ -Terme können auch auf der linken Seite vorkommen.
- In diesem Fall wird eine **Inklusionsbeziehung** ausgedrückt.
- So steht $n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$ für $\{n^2 + f \mid f \in \mathcal{O}(n)\} \subseteq \mathcal{O}(n^2)$.

Asymptotische Laufzeit und Landau-Notation

Beispiel

- $7 \log(n) + n^3 = \mathcal{O}(n^3)$ ist **richtig**.
- $7 \log(n)n^3 = \mathcal{O}(n^3)$ ist **falsch**.
- $2^{n+\mathcal{O}(1)} = \mathcal{O}(2^n)$ ist **richtig**.
- $2^{\mathcal{O}(n)} = \mathcal{O}(2^n)$ ist **falsch** (siehe Übungen).

Asymptotische Laufzeit und Landau-Notation

Satz

Es gilt:

$$f(n) = \mathcal{O}(g(n)) \iff 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

Beweis:

$$f(n) = \mathcal{O}(g(n)) \iff \exists c > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

$$\iff \exists c > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 : \frac{f(n)}{g(n)} < c$$

$$\iff 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

Asymptotische Laufzeit und Landau-Notation

Definition

- Weiter schreiben wir $f(n) = o(g(n))$, falls es für jedes $c > 0$ eine Zahl n_0 gibt mit

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Bedeutung: „ f wächst **wesentlich langsamer** als g .“

- Außerdem schreiben wir

- $f(n) = \Omega(g(n))$ für $g(n) = \mathcal{O}(f(n))$,

Bedeutung: „ f wächst **mindestens so schnell** wie g ,“

- $f(n) = \omega(g(n))$ für $g(n) = o(f(n))$,

Bedeutung: „ f wächst **wesentlich schneller** als g “ und

- $f(n) = \Theta(g(n))$ für $f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$.

Bedeutung: „ f und g wachsen **ungefähr gleich schnell**.“

Asymptotische Laufzeit und Landau-Notation

Satz: Außerdem gilt:

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

$$f(n) = \Omega(g(n)) \iff 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$$

$$f(n) = \omega(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) = \Theta(g(n)) \iff 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

Suchen von Mustern in Texten

Problem: String-Matching

Gegeben: Ein Text $x = x_1 \cdots x_n$ und ein Muster $y = y_1 \cdots y_m$ über einem Alphabet Σ .

Gesucht: Alle Vorkommen von y in x .

Bemerkung

- Wir sagen y kommt in x an Stelle i vor, falls $x_{i+1} \cdots x_{i+m} = y$ ist.
- Typische Anwendungen finden sich in Textverarbeitungssystemen (emacs, grep, etc.), sowie bei der DNS- bzw. DNA-Sequenzanalyse.

Suchen von Mustern in Texten

Beispiel

Sei $\Sigma = \{A,C,G,U\}$.

Text $x = \text{AUGACG}\text{AUGAUGUAGGUAGCGUAG}\text{AUGAUGUAG}$,

Muster $y = \text{AUGAUGUAG}$.

Das Muster y kommt im Text x an den Stellen **6** und **24** vor.

Naiver String-Matching Algorithmus

Algorithmus `naiv-String-Matcher(x, y)`

```
1  Input: Text  $x = x_1 \cdots x_n$  und Muster  $y = y_1 \cdots y_m$ 
2     $V := \emptyset$ 
3    for  $i := 0$  to  $n - m$  do
4      if  $x_{i+1} \cdots x_{i+m} = y_1 \cdots y_m$  then
5         $V := V \cup \{i\}$ 
6  Output:  $V$ 
```

Korrektheit von `naiv-String-Matcher`

- In der for-Schleife testet der Algorithmus alle potentiellen Stellen, an denen y in x vorkommen kann, und
- fügt in Zeile 5 genau die Stellen i zu V hinzu, für die $x_{i+1} \cdots x_{i+m} = y$ ist.

Naiver String-Matching Algorithmus

Algorithmus `naiv-String-Matcher(x, y)`

```
1  Input: Text  $x = x_1 \cdots x_n$  und Muster  $y = y_1 \cdots y_m$ 
2     $V := \emptyset$ 
3    for  $i := 0$  to  $n - m$  do
4      if  $x_{i+1} \cdots x_{i+m} = y_1 \cdots y_m$  then
5         $V := V \cup \{i\}$ 
6  Output:  $V$ 
```

Laufzeit von `naiv-String-Matcher`

- Die for-Schleife wird $(n - m + 1)$ -mal durchlaufen.
- Der Test in Zeile 4 benötigt maximal m Vergleiche.
- Dies führt auf eine Laufzeit von $\mathcal{O}(nm) = \mathcal{O}(n^2)$.
- Untere Schranke: Für Eingaben der Form $x = a^n$ und $y = a^{\lfloor n/2 \rfloor}$ ist die Laufzeit tatsächlich $\Theta(n^2)$.

String-Matching mit endlichen Automaten

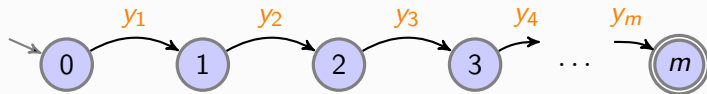
Idee

Konstruiere einen DFA M_y , der jedes Vorkommen von y in der Eingabe x durch Erreichen eines Endzustands anzeigt, d.h. M_y erkennt die Sprache

$$L = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}.$$

Konstruktion von M_y

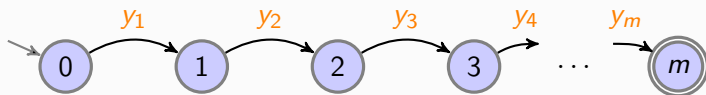
- M_y hat $m + 1$ Zustände, die den Präfixen $y_1 \cdots y_k$, $k = 0, \dots, m$, von $y = y_1 \cdots y_m$ entsprechen.
- Liest M_y im Zustand k das Zeichen y_{k+1} , so wechselt M_y in den Zustand $k + 1$, d.h. $\delta(k, y_{k+1}) = k + 1$ für $k = 0, \dots, m - 1$:



String-Matching mit endlichen Automaten

Konstruktion von M_y

- Liest M_y im Zustand k das Zeichen y_{k+1} , so wechselt M_y in den Zustand $k + 1$, d.h. $\delta(k, y_{k+1}) = k + 1$ für $k = 0, \dots, m - 1$:



- Falls das nächste Zeichen a nicht mit y_{k+1} übereinstimmt (engl. **mismatch**), wechselt M_y in den Zustand $\delta(k, a) = \max\{j \leq m \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}$.
- M_y speichert also in seinem Zustand die maximale Länge k eines Präfixes $y_1 \cdots y_k$ von y , das Suffix der gelesenen Eingabe ist: $\hat{\delta}(0, x) = \max\{k \leq m \mid y_1 \cdots y_k \text{ ist Suffix von } x\}$.

String-Matching mit endlichen Automaten

Konstruktion von M_y

- M_y speichert also in seinem Zustand die maximale Länge k eines Präfixes $y_1 \cdots y_k$ von y , das Suffix der gelesenen Eingabe ist:
$$\hat{\delta}(0, x) = \max\{j \leq m \mid y_1 \cdots y_j \text{ ist Suffix von } x\}.$$
- Die Korrektheit von M_y folgt aus der Beobachtung, dass M_y isomorph zum Äquivalenzklassen-DFA M_{R_L} für L ist.
- M_{R_L} hat die Zustände $[y_1 \cdots y_k]$, $k = 0, \dots, m$, von denen nur $[y_1 \cdots y_m]$ ein Endzustand ist.
- Die Überföhrungsfunktion ist definiert durch

$$\delta([y_1 \cdots y_k], a) = [y_1 \cdots y_j],$$

wobei $y_1 \cdots y_j$ das längste Präfix von y ist, welches Suffix von $y_1 \cdots y_k a$ ist (siehe folgenden Satz).

String-Matching mit endlichen Automaten

Satz: Sei $L = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}$.

Der Äquivalenzklassen DFA M_{R_L} hat die Zustände $[y_1 \cdots y_k]$, $k = 0, \dots, m$

Beweis: Zur Erinnerung: $x_1 R_L x_2 \iff L_{x_1} = L_{x_2}$, wobei $L_x = \{z \in \Sigma^* \mid xz \in L\}$. In unserem Fall ist $L_x = \{z \mid y \text{ ist Suffix von } xz\}$. Es ist zweierlei zu zeigen:

1. $\forall x \in \Sigma^* : \exists k : L_x = L_{y_1 \cdots y_k}$
2. $\forall j, k \leq m, j \neq k : L_{y_1 \cdots y_j} \neq L_{y_1 \cdots y_k}$

String-Matching mit endlichen Automaten

1. Es ist zu zeigen: $\forall x \in \Sigma^* : \exists k : L_x = L_{y_1 \dots y_k}$

Sei $x \in \Sigma^*$ beliebig, $L_x = \{z \mid xz \in L\} = \{z \mid y \text{ ist Suffix von } xz\}$.

Wähle $k := \max\{j \leq m \mid y_1 \dots y_j \text{ ist Suffix von } x\}$.

Zu zeigen $L_x = L_{y_1 \dots y_k}$. Es gilt: $y_1 \dots y_k$ ist ein Suffix von x .

$\supseteq$:

$$z \in L_{y_1 \dots y_k} \Rightarrow \underbrace{y_1 \dots y_k}_{\text{Suffix von } x} z \in L \Rightarrow xz \in L \Rightarrow z \in L_x$$

$\subseteq$: $z \in L_x \Rightarrow xz \in L \Rightarrow y \text{ Suffix von } xz \Rightarrow$
 $|z| \geq |y|$ oder x endet mit einem Präfix von y .

Fall $|z| \geq |y|$: $z \in L = L_{y_1 \dots y_m} = L_y$

Fall x endet mit einem Präfix von y : $xz = \dots x_{n-k} \underbrace{y_1 \dots y_j}_z z \in L$

$y_1 \dots y_k$ das längste solche Präfix von y das zugleich Suffix von x
 $\Rightarrow y \text{ Suffix von } y_1 \dots y_k z \Rightarrow y_1 \dots y_k z \in L \Rightarrow z \in L_{y_1 \dots y_k}$.

String-Matching mit endlichen Automaten

2. Es ist zu zeigen: $\forall j, k \leq m, j \neq k : L_{y_1 \dots y_j} \neq L_{y_1 \dots y_k}$

Sei $j < k \leq m$. Dann ist $y_{k+1} \dots y_m \in L_{y_1 \dots y_k}$, kann aber nicht in $L_{y_1 \dots y_j}$ sein, da $|y_1 \dots y_j y_{k+1} \dots y_m| < m$.

Satz: $\hat{\delta}(z_0, x) = z_k$ mit $k = \max\{j \leq m \mid y_1 \dots y_j \text{ ist Suffix von } x\}$.

Zur Erinnerung: $\delta(z_k, a) = z_j$ mit

$j = \max\{j \leq m \mid y_1 \dots y_j \text{ ist Suffix von } y_1 \dots y_k a\}$.

Beweis des Satzes durch Induktion über $|x|$.

IA. $|x| = 0$: $x = \varepsilon$: $\hat{\delta}(z_0, x) = \hat{\delta}(z_0, \varepsilon) = z_0$

IS. Sei $x = x_1 \dots x_n$. Dann

$$\hat{\delta}(z_0, x) = \hat{\delta}(z_0, x_1 \dots x_n) = \delta(\underbrace{\hat{\delta}(z_0, x_1 \dots x_{n-1})}_{z_{k'}}, x_n)$$

String-Matching mit endlichen Automaten

$$\begin{aligned}\hat{\delta}(z_0, x) &= \hat{\delta}(z_0, x_1 \cdots x_n) = \delta(\underbrace{\hat{\delta}(z_0, x_1 \cdots x_{n-1})}_{z_{k'}}, x_n) \\ &= \begin{cases} z_{k'+1} & \text{falls } y_1 \cdots y_{k'+1} \text{ ist Suffix von } x_1 \cdots x_n \\ z_k & \text{falls } k = \max\{j \leq m \mid \underbrace{y_1 \cdots y_j}_{\text{ist Suffix von } y_1 \cdots y_{k'} x_n}\} \end{cases} \\ &= \begin{cases} z_{k'+1} & \text{falls } x_n = y_{k'+1} \\ z_k & \text{falls } k = \max\{j \leq m \mid \underbrace{y_1 \cdots y_{k'}}_{\text{Suff}(x_1 \cdots x_{n-1})} x_n \\ & \text{ist Suffix von } x\} \end{cases}\end{aligned}$$

$$k' = \max\{j \leq m \mid y_1 \cdots y_j \text{ ist Suffix von } x_1 \cdots x_{n-1}\}$$

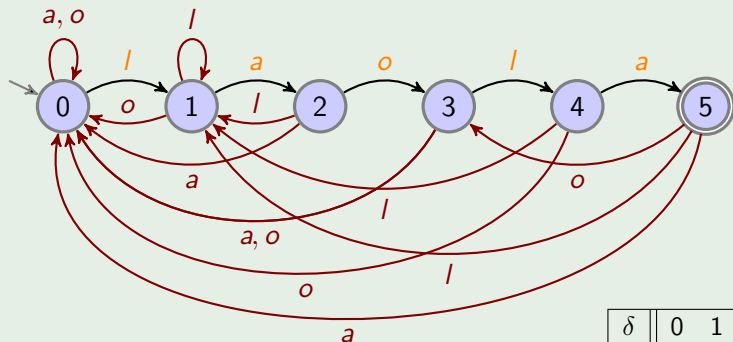
$$k = \max\{j \leq m \mid y_1 \cdots y_j \text{ ist Suffix von } x = x_1 \cdots x_{n-1} x_n\}$$

$y_1 \cdots y_j$ ist das längste Präfix von y , das auch Suffix von x .

String-Matching mit endlichen Automaten

Beispiel

Für das Muster $y = laola$ ergibt sich folgender DFA M_y :

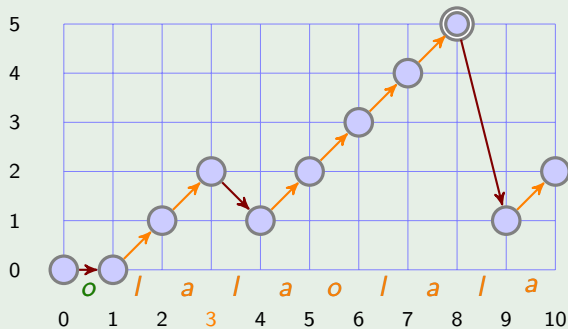


δ	0	1	2	3	4	5
a	0	2	0	0	5	0
l	1	1	1	4	1	1
o	0	0	3	0	0	3

String-Matching mit endlichen Automaten

Beispiel

M_y macht bei der Suche nach dem Muster $y = laola$ im Text $x = olalaolala$ folgende Übergänge:



δ	0	1	2	3	4	5
a	0	2	0	0	5	0
l	1	1	1	4	1	1
o	0	0	3	0	0	3

String-Matching mit endlichen Automaten

Algorithmus DFA-String-Matcher(x, y)

```
1  Input: Text  $x = x_1 \cdots x_n$  und Muster  $y = y_1 \cdots y_m$ 
2      konstruiere den DFA  $M_y = (Z, \Sigma, \delta, 0, m)$  mit  $Z = \{0, \dots, m\}$ 
3       $V := \emptyset$ 
4       $k := 0$ 
5      for  $i := 1$  to  $n$  do
6           $k := \delta(k, x_i)$ 
7          if  $k = m$  then  $V := V \cup \{i - m\}$ 
8  Output:  $V$ 
```

Korrektheit von DFA-String-Matcher

Wegen

$$L(M_y) = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}.$$

fügt der Algorithmus genau die Stellen $j = i - m$ zu V hinzu, für die y ein Suffix von $x_1 \cdots x_i$ (also $x_{j+1} \cdots x_{j+m} = y$) ist.

String-Matching mit endlichen Automaten

Laufzeit von DFA-String-Matcher

- Die Laufzeit ist die Summe der Laufzeiten für die Konstruktion von M_y und für die Simulation von M_y bei Eingabe x , wobei letztere durch $\mathcal{O}(n)$ beschränkt ist.
- Für δ ist eine Tabelle mit $(m+1)\|\Sigma\|$ Einträgen
$$\delta(k, a) = \max\{j \leq k+1 \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}$$
zu berechnen.
- Jeder Eintrag $\delta(k, a)$ ist in Zeit $\mathcal{O}(k^2) = \mathcal{O}(m^2)$ berechenbar.
- Dies führt auf eine Laufzeit von $\mathcal{O}(\|\Sigma\| m^3)$ für die Konstruktion von M_y und somit auf eine Gesamtlaufzeit von $\mathcal{O}(\|\Sigma\| m^3 + n)$.
- Tatsächlich lässt sich M_y sogar in Zeit $\mathcal{O}(\|\Sigma\| m)$ konstruieren (mit der Präfixfunktion π , siehe unten, siehe Übungen).

Der Knuth-Morris-Pratt-Algorithmus

- Tritt im Zustand k ein Mismatch $a \neq y_{k+1}$ auf, so ermittelt M_y das längste Präfix p von $y_1 \cdots y_k$, das zugleich Suffix von $y_1 \cdots y_k a$ ist, und springt in den Zustand $j = \delta(k, a) = |p|$.
- Im Fall $j > 0$ hat p also die Form $p = p' a$, wobei $p' = y_1 \cdots y_{j-1}$ sowohl echtes Präfix als auch echtes Suffix von $y_1 \cdots y_k$ ist. Zudem gilt $y_j = a$.
- Die Idee beim KMP-Algorithmus ist nun, bei einem Mismatch unabhängig von a auf das nächst kleinere Präfix $\tilde{p} = y_1 \cdots y_i$ von $y_1 \cdots y_k$ zu springen, das auch Suffix von $y_1 \cdots y_k$ ist.
- Stimmt nach diesem Rücksprung das nächste Eingabezeichen a mit y_{i+1} überein, so wird dieses gelesen und der KMP-Algorithmus erreicht (nach einem kleinen Umweg über den Zustand i) den Zustand $i + 1 = j$, in den auch M_y wechselt.

Der Knuth-Morris-Pratt-Algorithmus

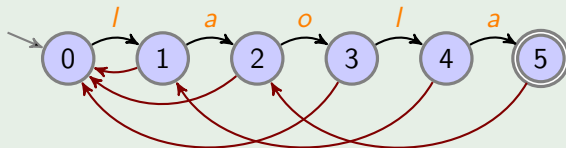
- Stimmt nach diesem Rücksprung das nächste Eingabezeichen a mit y_{i+1} überein, so wird dieses gelesen und der KMP-Algorithmus erreicht (nach einem kleinen Umweg über den Zustand i) den Zustand $i + 1 = j$, in den auch M_y wechselt.
- Andernfalls springt der KMP-Algorithmus nach derselben Methode solange weiter zurück, bis das nächste Eingabezeichen a „passt“ (also $y_{i+1} = a$ und somit $\tilde{p}a$ ein Präfix von y ist) oder der Zustand 0 erreicht wird.
- In beiden Fällen wird a gelesen und der Zustand $\delta(k, a)$ angenommen.
- Der KMP-Algorithmus besucht also alle Zustände, die auch M_y besucht, führt aber die Rücksprünge in mehreren Etappen aus.

Der Knuth-Morris-Pratt-Algorithmus

- Die Sprungadressen werden durch die so genannte **Präfixfunktion** $\pi : \{1, \dots, m\} \rightarrow \{0, \dots, m-1\}$ ermittelt:
$$\pi(k) = \max\{j \geq 0 \mid y_1 \cdots y_j \text{ ist echtes Suffix von } y_1 \cdots y_k\}.$$

Beispiel

Für das Muster $y = laola$ ergibt sich folgende Präfixfunktion π :

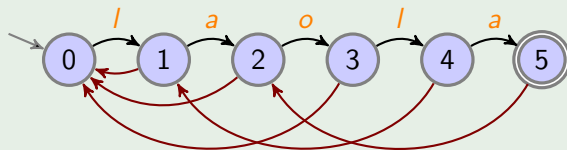


k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

Der Knuth-Morris-Pratt-Algorithmus

Beispiel

Für das Muster $y = \text{laola}$ ergibt sich folgende Präfixfunktion π :



k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2



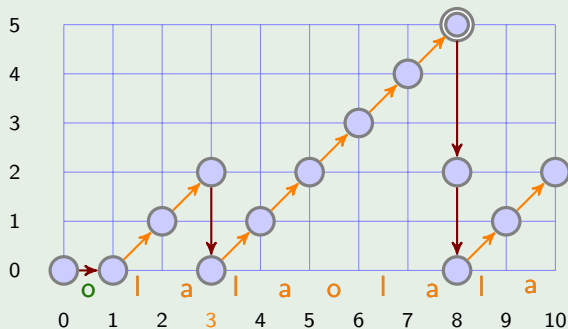
Wir können uns die Arbeitsweise dieses Automaten wie folgt vorstellen:

- 1 Erlaubt das nächste Eingabezeichen einen Übergang vom aktuellen Zustand k nach $k + 1$, so führe diesen aus.
- 2 Ist ein Übergang nach $k + 1$ nicht möglich und $k \geq 1$, so springe in den Zustand $\pi(k)$ ohne das nächste Zeichen zu lesen.
- 3 Andernfalls (d.h. $k = 0$ und ein Übergang nach 1 ist nicht möglich) lies das nächste Zeichen und bleibe im Zustand 0.

Der Knuth-Morris-Pratt-Algorithmus

Beispiel

Der KMP-Algorithmus macht bei der Suche nach dem Muster $y = \text{laola}$ im Text $x = \text{olalaolala}$ folgende Übergänge:



k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

Der Knuth-Morris-Pratt-Algorithmus

Nun können wir den Algorithmus DFA-String-Matcher in den KMP-Algorithmus umwandeln:

DFA-String-Matcher(x, y)

```
1  Input: Text  $x = \dots x_n$  und  
    Muster  $y = y_1 \dots y_m$   
2  konstruiere den DFA  $M_y$   
3   $V := \emptyset$   
4   $k := 0$   
5  for  $i := 1$  to  $n$  do  
6     $k := \delta(k, x_i)$   
  
7    if  $k = m$  then  
       $V := V \cup \{i - m\}$   
8  Output:  $V$ 
```

KMP-String-Matcher(x, y)

```
1  Input: Text  $x = x_1 \dots x_n$  und  
    Muster  $y = y_1 \dots y_m$   
2   $\pi := \text{KMP-Prefix}(y)$   
3   $V := \emptyset$   
4   $k := 0$   
5  for  $i := 1$  to  $n$  do  
6    while ( $k > 0 \wedge x_i \neq y_{k+1}$ ) do  
7       $k := \pi(k)$   
8    if  $x_i = y_{k+1}$  then  $k := k + 1$   
9    if  $k = m$  then  
10      $V := V \cup \{i - m\}$ ,  $k := \pi(k)$   
11 Output:  $V$ 
```

Der Knuth-Morris-Pratt-Algorithmus

Korrektheit von KMP-String-Matcher

- Da KMP-String-Matcher den Zustand m an genau den gleichen Textstellen annimmt wie DFA-String-Matcher, findet er wie dieser alle Vorkommen von y .

Laufzeit von KMP-String-Matcher (ohne KMP-Prefix)

- Die Laufzeit ist proportional zur Anzahl der Zustandsübergänge.
- Bei jedem Schritt wird der Zustand um maximal Eins erhöht.
- Daher kann der Zustand nicht öfter verkleinert werden als er erhöht wird (**Amortisationsanalyse**).
- Es gibt genau n Zustandsübergänge, bei denen der Zustand erhöht wird bzw. unverändert bleibt.
- Insgesamt finden also höchstens $2n = \mathcal{O}(n)$ Zustandsübergänge statt.

Berechnung der KMP-Präfixfunktion

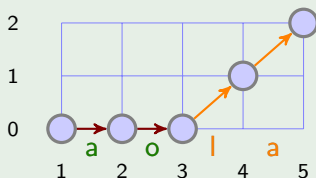
- Für die Berechnung von π benötigen wir für jedes Präfix $y_1 \cdots y_i$ das längste echte Präfix, das zugleich Suffix von $y_1 \cdots y_i$ ist.
- Die Idee besteht nun darin, mit dem KMP-Algorithmus das Muster y im Text $y_2 \cdots y_m$ zu suchen.
- Dann liefert der beim Lesen von y_i erreichte Zustand k gerade das längste Präfix $y_1 \cdots y_k$, das zugleich Suffix von $y_2 \cdots y_i$ ist (d.h. es gilt $\pi(i) = k$).
- Zudem werden bis zum Lesen von y_i nur Zustände kleiner als i angenommen.
- Daher sind die π -Werte für alle bis dahin auszuführenden Rücksprünge bereits bekannt und π kann in Zeit $\mathcal{O}(m)$ berechnet werden.

Berechnung der KMP-Präfixfunktion

Prozedur KMP-Prefix(y)

```
1  Input: Muster  $y = y_1 \cdots y_m$   
2   $\pi(1) := 0$   
3   $k := 0$   
4  for  $i := 2$  to  $m$  do  
5      while ( $k > 0 \wedge y_i \neq y_{k+1}$ ) do  $k := \pi(k)$   
6      if  $y_i = y_{k+1}$  then  $k := k + 1$   
7       $\pi(i) := k$   
8  Output:  $\pi$ 
```

Beispiel: Ablaufprotokoll von KMP-Prefix bei Eingabe $y = \text{laola}$



i	1	2	3	4	5
$\pi(i)$	0	0	0	1	2

Vergleich der betrachteten Algorithmen

Algorithmus	Vorverarbeitung	Suche	Gesamtlaufzeit
naiv	0	$\mathcal{O}(nm)$	$\mathcal{O}(nm)$
DFA (einfach)	$\mathcal{O}(\ \Sigma\ m^3)$	$\mathcal{O}(n)$	$\mathcal{O}(\ \Sigma\ m^3 + n)$
DFA (verbessert)	$\mathcal{O}(\ \Sigma\ m)$	$\mathcal{O}(n)$	$\mathcal{O}(\ \Sigma\ m + n)$
Knuth-Morris-Pratt	$\mathcal{O}(m)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Durchsuchen von Zahlenfolgen

Problem: Element-Suche

Gegeben: Eine Folge $a_1, \dots, a_n$ von natürlichen Zahlen und eine Zahl a .

Gesucht: Ein Index i mit $a_i = a$ (bzw. eine Fehlermeldung, falls $a \notin \{a_1, \dots, a_n\}$ ist).

Bemerkung

- Typische Anwendungen finden sich bei der Verwaltung von Datensätzen, wobei jeder Datensatz über einen eindeutigen Schlüssel (z.B. **Matrikelnummer**) zugreifbar ist.
- Bei manchen Anwendungen können die Zahlen in der Folge auch mehrfach vorkommen (in diesem Fall ist $\{a_1, \dots, a_n\}$ eine **Multimenge**). Gesucht sind dann evtl. alle Indizes i mit $a_i = a$.

Durchsuchen von Zahlenfolgen

Durch eine sequentielle Suche lässt sich das Problem in Zeit $\mathcal{O}(n)$ lösen.

Algorithmus Sequential-Search

```
1  Input: Eine Zahlenfolge  $a_1, \dots, a_n$  und eine Zahl  $a$ 
2       $i := 0$ 
3      repeat
4           $i := i + 1$ 
5      until  $(i = n \vee a = a_i)$ 
6  Output:  $i$ , falls  $a_i = a$  bzw. Fehlermeldung, falls  $a_i \neq a$ 
```

Suche in sortierten Folgen

Falls die Folge $a_1, \dots, a_n$ sortiert ist, d.h. es gilt $a_i \leq a_j$ für $i \leq j$, bietet sich eine **Binärsuche** an.

Algorithmus Binary-Search

```
1  Input: Eine sortierte Folge  $a_1, \dots, a_n$  und eine Zahl  $a$ 
2       $l := 1$ 
3       $r := n$ 
4      while  $l < r$  do
5           $m := \lfloor (l + r) / 2 \rfloor$ 
6          if  $a \leq a_m$  then  $r := m$  else  $l := m + 1$ 
7  Output:  $l$ , falls  $a_l = a$  bzw. Fehlermeldung, falls  $a_l \neq a$ 
```

Suche in sortierten Folgen

Algorithmus Binary-Search

```
1  Input: Eine sortierte Folge  $a_1, \dots, a_n$  und eine Zahl  $a$ 
2       $l := 1$ 
3       $r := n$ 
4      while  $l < r$  do
5           $m := \lfloor (l + r) / 2 \rfloor$ 
6          if  $a \leq a_m$  then  $r := m$  else  $l := m + 1$ 
7  Output:  $l$ , falls  $a_l = a$  bzw. Fehlermeldung, falls  $a_l \neq a$ 
```

Korrektheit von Binary-Search

- Offensichtlich gibt der Algorithmus im Fall $a \notin \{a_1, \dots, a_n\}$ eine Fehlermeldung aus (falls er überhaupt stoppt).
- Im Fall $a \in \{a_1, \dots, a_n\}$ gilt die **Schleifeninvariante** $l \leq r \wedge a_l \leq a \leq a_r$.
- Da nach Abbruch der while-Schleife zudem $l \geq r$ gilt, muss dann $l = r$ und $a = a_l$ sein.

Suche in sortierten Folgen

Algorithmus Binary-Search

```
1  Input: Eine sortierte Folge  $a_1, \dots, a_n$  und eine Zahl  $a$ 
2       $l := 1$ 
3       $r := n$ 
4      while  $l < r$  do
5           $m := \lfloor (l + r)/2 \rfloor$ 
6          if  $a \leq a_m$  then  $r := m$  else  $l := m + 1$ 
7  Output:  $l$ , falls  $a_l = a$  bzw. Fehlermeldung, falls  $a_l \neq a$ 
```

Laufzeit von Binary-Search

- Die Länge $l - r + 1$ des Suchintervalls $[l, r]$ wird in jedem Schleifendurchlauf auf $\lfloor (l - r + 1)/2 \rfloor$ oder auf $\lceil (l - r + 1)/2 \rceil$ reduziert.
- Daher werden höchstens $\lceil \log_2 n \rceil$ Schleifendurchläufe ausgeführt, d.h. die Laufzeit ist $\mathcal{O}(\log n)$.

Sortieren von Zahlenfolgen

Sortierproblem

Gegeben: Eine Folge $a_1, \dots, a_n$ von natürlichen Zahlen.

Gesucht: Eine Permutation $a_{i_1}, \dots, a_{i_n}$ dieser Folge mit $a_{i_j} \leq a_{i_{j+1}}$ für $j = 1, \dots, n - 1$.

Bemerkung

- Man unterscheidet **vergleichende Sortierverfahren** von den übrigen Sortierverfahren.
- Während erstere nur Anfragen der Form $a_i \stackrel{?}{\leq} a_j$ bzw. $a_i \stackrel{?}{<} a_j$ stellen dürfen, können letztere auch die konkreten Zahlenwerte a_i der Folge abfragen.
- Erstere benötigen im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.
- Dagegen arbeiten letztere unter bestimmten Voraussetzungen sogar in Linearzeit.

Sortieren durch Einfügen

Wir können eine Zahlenfolge sortieren, indem wir sequentiell die Zahl a_i in die bereits sortierte Teilfolge $a_1, \dots, a_{i-1}$ einfügen.

Algorithmus InsertionSort

```
1  Input: Eine Zahlenfolge  $a_1, \dots, a_n$ 
2    for  $i := 2$  to  $n$  do
3       $z := a_i$ 
4       $j := i - 1$ 
5      while  $(j \geq 1 \wedge a_j > z)$  do
6         $a_{j+1} := a_j$ 
7         $j := j - 1$ 
8       $a_{j+1} := z$ 
9  Output:  $a_1, \dots, a_n$ 
```

Sortieren durch Einfügen

Algorithmus InsertionSort

<pre>1 Input: $a_1, \dots, a_n$ 2 for $i := 2$ to n do 3 $z := a_i$ 4 $j := i - 1$ 5 while $(j \geq 1 \wedge a_j > z)$ do</pre>	<pre>5 while $(j \geq 1 \wedge a_j > z)$ do 6 $a_{j+1} := a_j$ 7 $j := j - 1$ 8 $a_{j+1} := z$ 9 Output: $a_1, \dots, a_n$</pre>
---	---

Korrektheit von InsertionSort

Folgende Schleifeninvarianten lassen sich induktiv beweisen:

- Nach jedem Durchlauf der **for**-Schleife sind $a_1, \dots, a_i$ sortiert.
- Nach jedem Durchlauf der **while**-Schleife gilt: $a_{j+1}, \dots, a_{i-1}$ wurden um jeweils ein Feld nach $a_{j+2}, \dots, a_i$ verschoben und $z < a_{j+2}$.

Zusammen mit der Abbruchbedingung der **while**-Schleife ($j < 1 \vee a_j \leq z$) folgt hieraus, dass z in Zeile 8 an der richtigen Stelle eingefügt wird.

Sortieren durch Einfügen

Algorithmus InsertionSort

1	Input: $a_1, \dots, a_n$	5	while $(j \geq 1 \wedge a_j > z)$ do
2	for $i := 2$ to n do	6	$a_{j+1} := a_j$
3	$z := a_i$	7	$j := j - 1$
4	$j := i - 1$	8	$a_{j+1} := z$
5	while $(j \geq 1 \wedge a_j > z)$ do	9	Output: $a_1, \dots, a_n$

Laufzeit von InsertionSort

- Die while-Schleife wird für jedes $i = 2, \dots, n$ höchstens $(i - 1)$ -mal ausgeführt.
- Dies führt auf eine Laufzeit von $\sum_{i=2}^n \mathcal{O}(i - 1) = \mathcal{O}(n^2)$.

Sortieren durch Einfügen

Bemerkung

- Ist die Eingabefolge $a_1, \dots, a_n$ bereits sortiert, so wird die while-Schleife niemals durchlaufen.
- Im **besten Fall** ist die Laufzeit daher $\sum_{i=2}^n \Theta(1) = \Theta(n)$.
- Ist die Eingabefolge $a_1, \dots, a_n$ dagegen absteigend sortiert, so wandert z in $i - 1$ Durchläufen der while-Schleife vom Ende an den Anfang der bereits sortierten Teilfolge $a_1, \dots, a_i$.
- Im **schlechtesten Fall** ist die Laufzeit also $\sum_{i=2}^n \Theta(i - 1) = \Theta(n^2)$.
- Bei einer zufälligen Eingabepermutation der Folge $1, \dots, n$ wird z im Erwartungswert in der Mitte der Teilfolge $a_1, \dots, a_i$ eingefügt.
- Folglich beträgt die (erwartete) Laufzeit im **durchschnittlichen Fall** ebenfalls $\sum_{i=2}^n \Theta(\frac{i-1}{2}) = \Theta(n^2)$.

Sortieren durch Mischen (Divide and Conquer)

Wir können eine Zahlenfolge sortieren, indem wir

- sie in zwei Teilfolgen zerlegen,
- diese durch rekursive Aufrufe sortieren und
- die sortierten Teilfolgen wieder zu einer Liste zusammenfügen.

Die “Teile und Herrsche”-Strategie

Die oben beschriebene Vorgehensweise ist unter dem Schlagwort “Divide and Conquer” (auch “divide et impera”, also “teile und herrsche”) bekannt. Dabei wird ein Problem gelöst, indem man es

- in mehrere Teilprobleme aufteilt,
- diese löst, und
- die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems zusammenfügt.

Sortieren durch Mischen

Die Prozedur $\text{MergeSort}(l, r)$ sortiert ein Feld $A[l \dots r]$, indem sie

- es in die Felder $A[l \dots m]$ und $A[m + 1 \dots r]$ zerlegt,
- diese durch jeweils einen rekursiven Aufruf sortiert, und
- die sortierten Teilfolgen durch einen Aufruf der Prozedur $\text{Merge}(l, m, r)$ zu einer sortierten Folge zusammenfügt.

Prozedur $\text{MergeSort}(l, r)$

```
1  if  $l < r$  then
2     $m := \lfloor (l + r) / 2 \rfloor$ 
3     $\text{MergeSort}(l, m)$ 
4     $\text{MergeSort}(m + 1, r)$ 
5     $\text{Merge}(l, m, r)$ 
```

Die Prozedur $\text{Merge}(l, m, r)$ mischt die beiden sortierten Felder $A[l \dots m]$ und $A[m + 1 \dots r]$ zu einem sortierten Feld $A[l \dots r]$.

Sortieren durch Mischen

Die Prozedur $\text{Merge}(l, m, r)$ mischt die beiden sortierten Felder $A[l \dots m]$ und $A[m + 1 \dots r]$ zu einem sortierten Feld $A[l \dots r]$.

Prozedur $\text{Merge}(l, m, r)$

<pre>1 allokiere Speicher für ein neues Feld $B[l \dots r]$ 2 $j := l$ 3 $k := m + 1$ 4 for $i := l$ to r do 5 if $j > m$ then 6 $B[i] := A[k]$ 7 $k := k + 1$ 8 else if $k > r$ then 9 $B[i] := A[j]$ 10 $j := j + 1$</pre>	<pre>10 $j := j + 1$ 11 else if $A[j] \leq A[k]$ then 12 $B[i] := A[j]$ 13 $j := j + 1$ 14 else 15 $B[i] := A[k]$ 16 $k := k + 1$ 17 kopiere das Feld $B[l \dots r]$ in das Feld $A[l \dots r]$ 18 gib den Speicher für B wieder frei</pre>
---	--

Sortieren durch Mischen

- Man beachte, dass Merge für die Zwischenspeicherung der gemischten Folge im Feld B zusätzlichen Speicher benötigt.
- MergeSort ist daher kein “in place”-Sortiervfahren, welches neben dem Speicherplatz für die Eingabefolge nur konstant viel zusätzlichen Speicher benötigt.
- Zum Beispiel ist InsertionSort ein “in place”-Verfahren.
- Auch MergeSort kann als ein “in place”-Sortiervfahren implementiert werden, falls die zu sortierende Zahlenfolge nicht als Array, sondern als mit Zeigern verkettete Liste vorliegt (siehe Übungen).

Sortieren durch Mischen

Korrektheit von MergeSort

Unter der Voraussetzung, dass Merge korrekt arbeitet, können wir per Induktion über die Länge $n = r - l + 1$ des zu sortierenden Arrays die Korrektheit von MergeSort beweisen:

$n = 1$: In diesem Fall tut MergeSort nichts, was offensichtlich korrekt ist.

$n \rightsquigarrow n + 1$: Um eine Folge der Länge $n + 1 \geq 2$ zu sortieren, zerlegt sie MergeSort in zwei Folgen der Länge höchstens n .

Diese werden

- durch die rekursiven Aufrufe nach IV korrekt sortiert und
- von Merge nach Voraussetzung korrekt zusammengefügt.

Sortieren durch Mischen

Korrektheit von Merge

Folgende Invariante der for-Schleife lässt sich leicht induktiv beweisen:

- Nach jeder Iteration enthält $B[l \dots i]$ die $i - l + 1$ kleinsten Werte aus $A[l \dots m]$ und $A[m + 1 \dots r]$ in sortierter Reihenfolge.
- Falls $j \leq m$ ist, gilt $B[i] \leq A[j]$, und falls $k \leq r$ ist, gilt $B[i] \leq A[k]$.

Nach der letzten Iteration (d.h. $i = r$) enthält daher $B[l \dots r]$ alle $r - l + 1$ Elemente aus $A[l \dots m]$ und $A[m + 1 \dots r]$ in sortierter Reihenfolge, womit die Korrektheit von Merge bewiesen ist.

Sortieren durch Mischen

Laufzeit von MergeSort

- Sei $V(n)$ die Anzahl der Vergleiche, die MergeSort im schlechtesten Fall benötigt, um ein Feld $A[l \dots r]$ der Länge $n = r - l + 1$ zu sortieren.
- Offensichtlich erfüllt $V(n)$ die Rekursionsgleichung

$$V(n) = \begin{cases} 0, & n = 1, \\ V(\lfloor n/2 \rfloor) + V(\lceil n/2 \rceil) + M(n), & \text{sonst.} \end{cases}$$

- Dabei ist $M(n) \leq n - 1$ die Anzahl der Vergleiche, die Merge benötigt, um die beiden sortierten Felder $A[l \dots m]$ und $A[m + 1 \dots r]$ zu mischen.
- Falls $M(n) = n - 1$ und n eine Zweierpotenz ist, erhalten wir also die Rekursion

$$V(n) = \begin{cases} 0, & n = 1, \\ 2V(n/2) + n - 1, & \text{sonst.} \end{cases}$$

- Für die Funktion $f(k) = V(2^k)$ gilt dann $f(0) = 0$ und
$$f(k) = 2f(k - 1) + 2^k - 1 \text{ für } k \geq 1.$$

Sortieren durch Mischen

Laufzeit von MergeSort

- Für die Funktion $f(k) = V(2^k)$ gilt dann $f(0) = 0$ und

$$f(k) = 2f(k-1) + 2^k - 1 \text{ für } k \geq 1.$$

- Aus den ersten Folgengliedern $f(0) = 0$, $f(1) = 1$,

$$f(2) = 2 + 2^2 - 1 = 1 \cdot 2^2 + 1,$$

$$f(3) = 2 \cdot 2^2 + 2 + 2^3 - 1 = 2 \cdot 2^3 + 1,$$

$$f(4) = 2 \cdot 2 \cdot 2^3 + 2 + 2^4 - 1 = 3 \cdot 2^4 + 1$$

lässt sich vermuten, dass $f(k) = (k-1) \cdot 2^k + 1$ ist.

- Dies lässt sich leicht durch Induktion über k verifizieren, woraus wir $V(n) = n \log_2 n - n + 1$ erhalten.
- Ist n eine beliebige Zahl und gilt nur $M(n) \leq n-1$, so folgt

$$V(n) \leq n' \log_2 n' - n' + 1 \leq 2n \log_2(2n) + 1 = \mathcal{O}(n \log n),$$

wobei $n' < 2n$ die kleinste Zweierpotenz $\geq n$ ist.

Sortieren durch Mischen

Da die Laufzeit $T(n)$ von MergeSort asymptotisch durch die Anzahl $V(n)$ der Vergleiche beschränkt ist, folgt $T(n) = \mathcal{O}(n \log n)$.

Satz

MergeSort ist ein vergleichendes Sortierverfahren mit einer Laufzeit von $\mathcal{O}(n \log n)$.

Weitere “Divide and Conquer”-Verfahren

- Wir werden noch weitere “Divide and Conquer”-Verfahren kennenlernen.
- Wie wir gesehen haben, besteht die Strategie darin, ein Problem der Größe n dadurch zu lösen, dass man es
 - in ℓ Teilprobleme der Größen $n_1, \dots, n_\ell$ aufteilt,
 - diese Teilprobleme löst, und
 - die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems zusammenfügt.

Laufzeitabschätzung bei “Divide and Conquer”-Verfahren

Dieser Ansatz ergibt typischerweise

- einfache Implementierungen,
- einfache Korrektheitsbeweise und
- Rekursionsgleichungen für die Laufzeitabschätzung der Form

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n \text{ „klein“ ist,} \\ D(n) + \sum_{i=1}^{\ell} T(n_i) + C(n), & \text{sonst.} \end{cases}$$

- Dabei ist $T(n)$ der Gesamtaufwand, $D(n)$ der Aufwand für das Teilen und $C(n)$ der Aufwand für das Zusammenfügen der Teillösungen.

Lösen/Abschätzen von Rekursionsgleichungen

Wie lassen sich Rekursionsgleichungen lösen?

- Lösung „raten“ und per Induktion beweisen.
- Lösung mit **Rekursionsbäumen**: „gezielt raten“ und dann per Induktion verifizieren (siehe **Übungen**).
- Das **Mastertheorem** (Hauptsatz der Laufzeitfunktionen, Satz von Akra & Bazzi) anwenden.

Das Mastertheorem

Mastertheorem (Akra & Bazzi)

- Sei $T : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion der Form

$$T(n) = \sum_{i=1}^{\ell} T(n_i) + f(n) \quad \text{mit} \quad n_i \in \{\lfloor \alpha_i n \rfloor, \lceil \alpha_i n \rceil\},$$

wobei $0 < \alpha_i < 1$, $i = 1, \dots, \ell$, fest gewählte reelle Zahlen sind.

- Dann gilt im Fall $f(n) = \Theta(n^k)$ für ein $k \geq 0$:

$$T(n) = \begin{cases} \Theta(n^k), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k < 1, \\ \Theta(n^k \log n), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k = 1, \\ \Theta(n^c), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k > 1, \end{cases}$$

wobei c (die eindeutige) Lösung der Gleichung $\sum_{i=1}^{\ell} \alpha_i^c = 1$ ist.

Das Mastertheorem

Mastertheorem (Akra & Bazzi)

Seien $0 < \alpha_i < 1$, $i = 1, \dots, \ell$ reelle Zahlen und $T : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion der Form

$$T(n) = \sum_{i=1}^{\ell} T(n_i) + f(n) \quad \text{mit} \quad n_i \in \{\lfloor \alpha_i n \rfloor, \lceil \alpha_i n \rceil\} \quad \text{und} \quad f(n) = \Theta(n^k).$$

Dann gilt

$$T(n) = \begin{cases} \Theta(n^k), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k < 1, \\ \Theta(n^k \log n), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k = 1, \\ \Theta(n^c), & \text{falls } \sum_{i=1}^{\ell} \alpha_i^k > 1, \sum_{i=1}^{\ell} \alpha_i^c = 1. \end{cases}$$

Beispiel

- Die Anzahl $V(n)$ der Vergleiche von MergeSort erfüllt die Rekursion

$$V(n) = V(\lfloor n/2 \rfloor) + V(\lceil n/2 \rceil) + n - 1,$$

d.h. $\ell = 2$, $\alpha_1 = \alpha_2 = 1/2$ und $f(n) = n - 1 = \Theta(n^k)$ für $k = 1$.

- Wegen $\sum_{i=1}^{\ell} \alpha_i^k = 1/2 + 1/2 = 1$ folgt daher $V(n) = \Theta(n \log n)$.

Eine untere Schranke für vergleichende Sortialgorithmen

Frage

Wie viele Fragen benötigt ein vergleichender Sortialgorithmus A mindestens, um eine Folge $(a_1, \dots, a_n)$ von n Zahlen zu sortieren?

Antwort

- Wir betrachten alle $n!$ Eingabefolgen $(a_1, \dots, a_n)$ der Form $(\pi(1), \dots, \pi(n))$, wobei $\pi \in S_n$ eine beliebige Permutation auf der Menge $\{1, \dots, n\}$ ist.

Eine untere Schranke für Sortieralgorithmen

Antwort

- Wir betrachten alle $n!$ Eingabefolgen $(a_1, \dots, a_n)$ der Form $(\pi(1), \dots, \pi(n))$, wobei $\pi \in S_n$ eine beliebige Permutation ist.
- Um diese Folgen korrekt zu sortieren, muss A solange Fragen der Form $a_i < a_j$ (bzw. $\pi(i) < \pi(j)$) stellen, bis höchstens noch eine Permutation $\pi \in S_n$ mit den erhaltenen Antworten konsistent ist.
- Wir beantworten jede Frage von A so, dass mindestens die Hälfte der verbliebenen Permutationen mit der gegebenen Antwort konsistent ist (**Mehrheitsvotum**).
- Folglich sind mindestens $n!/2^i$ Permutationen mit den Antworten auf die ersten i Fragen konsistent.
- Wegen $n! \sim \sqrt{2\pi n}(n/e)^n$ benötigt A also mindestens

$$\lceil \log_2(n!) \rceil = n \log_2 n - n \log_2 e + \frac{1}{2} \log n + \Theta(1) = n \log_2 n - \Theta(n)$$

Fragen, um die Anzahl der konsistenten Permutationen von $n!$ auf Eins zu reduzieren.

Eine untere Schranke für Sortialgorithmen

Veranschaulichung des Beweisarguments durch einen Fragebaum

- Wir können das Verhalten von A wie folgt durch einen Fragebaum B modellieren.
- Die Wurzel von B ist mit der ersten Frage von A markiert.
- Jeder mit einer Frage markierte Knoten hat zwei Kinder, die die Antworten ja und nein auf diese Frage repräsentieren.
- Stellt A nach Erhalt der Antwort eine weitere Frage, so markieren wir den entsprechenden Antwortknoten mit dieser Frage.
- Andernfalls gibt A eine Permutation π der Eingabefolge aus und der zugehörige Antwortknoten ist ein mit π markiertes Blatt.
- Nun ist leicht zu sehen, dass die Tiefe von B mit der Anzahl der von A benötigten Fragen im schlechtesten Fall übereinstimmt.
- Da jede Eingabefolge zu einem anderen Blatt führen muss (sonst ist A nicht korrekt), hat B mindestens $n!$ Blätter und damit mindestens die Tiefe $\lceil \log_2(n!) \rceil$.

Fazit: MergeSort ist asymptotisch optimal.

Da jedes vergleichende Sortierverfahren mindestens $\Omega(n \log n)$ Fragen benötigt, ist MergeSort asymptotisch optimal.

Korollar

MergeSort ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.

QuickSort

- QuickSort arbeitet ebenfalls nach der “Divide and Conquer”-Strategie.
- Bei QuickSort fällt jedoch die Hauptarbeit für die Zerlegung in Teilprobleme an, während das Zusammenfügen der Teillösungen trivial ist.
- Bei MergeSort ist es gerade umgekehrt.

QuickSort

Die Prozedur $\text{QuickSort}(l, r)$ sortiert ein Feld $A[l \dots r]$ wie folgt:

- Zuerst wird die Funktion $\text{Partition}(l, r)$ aufgerufen.
- Diese wählt ein **Pivotelement**, welches sich nach dem Aufruf in $A[m]$ befindet, und sortiert das Feld so um, dass gilt:

$$A[i] \leq A[m] \leq A[j] \text{ für alle } i, j \text{ mit } l \leq i < m < j \leq r.$$

- Danach werden die beiden Teilfolgen $A[l \dots m - 1]$ und $A[m + 1 \dots r]$ durch jeweils einen rekursiven Aufruf sortiert.

Prozedur $\text{QuickSort}(l, r)$

```
1  if  $l < r$  then  $m := \text{Partition}(l, r)$ 
2      QuickSort( $l, m - 1$ )
3      QuickSort( $m + 1, r$ )
```

QuickSort

Die Funktion $\text{Partition}(l, r)$ **pivotisiert** das Feld $A[l \dots r]$, indem sie

- $x = A[r]$ als Pivotelement wählt,
- die übrigen Elemente mit x vergleicht und dabei umsortiert und
- den neuen Index von x im umsortierten Feld A zurückgibt.

Prozedur $\text{Partition}(l, r)$

```
1   $i := l - 1$ 
2  for  $j := l$  to  $r - 1$  do
3      if  $A[j] \leq A[r]$  then
4           $i := i + 1$ 
5          if  $i < j$  then
6              vertausche  $A[i]$  und  $A[j]$ 
7  if  $i + 1 < r$  then
8      vertausche  $A[i + 1]$  und  $A[r]$ 
9  return( $i + 1$ )
```

QuickSort

Prozedur QuickSort(l, r)

```
1  if  $l < r$  then  $m := \text{Partition}(l, r)$   
2    QuickSort( $l, m - 1$ )  
3    QuickSort( $m + 1, r$ )
```

Korrektheit von QuickSort

Unter der Voraussetzung, dass die Funktion Partition korrekt arbeitet, d.h. nach ihrem Aufruf gilt

$$A[i] \leq A[m] \leq A[j] \text{ für alle } i, j \text{ mit } l \leq i < m < j \leq r,$$

folgt die Korrektheit von QuickSort durch einen einfachen Induktionsbeweis über die Länge $n = r - l + 1$ des zu sortierenden Arrays.

Korrektheit von Partition

Prozedur Partition(l, r)

```
1   $i := l - 1$ 
2  for  $j := l$  to  $r - 1$  do
3      if  $A[j] \leq A[r]$  then
4           $i := i + 1$ 
5          if  $i < j$  then
6              vertausche  $A[i]$  und  $A[j]$ 
7  if  $i + 1 < r$  then
8      vertausche  $A[i + 1]$  und  $A[r]$ 
9  return( $i + 1$ )
```

Die Korrektheit von Partition folgt leicht aus folgender Invariante

$A[k] \leq A[r]$ für $k = l, \dots, i$ und $A[k] > A[r]$ für $k = i + 1, \dots, j$.

Da nämlich nach Ende der for-Schleife $j = r - 1$ ist, garantiert die Vertauschung von $A[i + 1]$ und $A[r]$ die Korrektheit von Partition.

Korrektheit von Partition

Um eindeutig definierte Werte von j vor und nach jeder Iteration der for-Schleife zu haben, ersetzen wir sie durch eine while-Schleife.

```
1   $i := l - 1$ 
2  for  $j := l$  to  $r - 1$  do
    3      if  $A[j] \leq A[r]$  then
    4           $i := i + 1$ 
    5      if  $i < j$  then vertausche
    6           $A[i]$  und  $A[j]$ 
```

```
1   $i := l - 1$ ;  $j := l - 1$ 
2  while  $j < r - 1$  do
    3       $j := j + 1$ 
    4      if  $A[j] \leq A[r]$  then
    5           $i := i + 1$ 
    6      if  $i < j$  then vertausche
    7           $A[i]$  und  $A[j]$ 
```

Beweis der Schleifeninvariante

$A[k] \leq A[r]$ für $k = l, \dots, i$ und $A[k] > A[r]$ für $k = i + 1, \dots, j$.

IA: Vor Beginn der while-Schleife gilt die Invariante, da i und j den Wert $l - 1$ haben.

Korrektheit von Partition

```
1   $i := l - 1$ 
2  for  $j := l$  to  $r - 1$  do
3      if  $A[j] \leq A[r]$  then
4           $i := i + 1$ 
5      if  $i < j$  then vertausche
6           $A[i]$  und  $A[j]$ 
```

```
1   $i := l - 1$ ;  $j := l - 1$ 
2  while  $j < r - 1$  do
3       $j := j + 1$ 
4      if  $A[j] \leq A[r]$  then
5           $i := i + 1$ 
6      if  $i < j$  then vertausche
7           $A[i]$  und  $A[j]$ 
```

Beweis der Schleifeninvariante

$A[k] \leq A[r]$ für $k = l, \dots, i$ und $A[k] > A[r]$ für $k = i + 1, \dots, j$.

IS: Zunächst wird j hochgezählt und dann $A[j]$ mit $A[r]$ verglichen.

$A[j] > A[r]$: Es ist klar, dass die Invariante ihre Gültigkeit behält, da nur j hochgezählt wird und i sich nicht ändert.

$A[j] \leq A[r]$: In diesem Fall wird auch i hochgezählt.
Im Fall $i = j$ gilt dann $A[i] = A[j] \leq A[r]$.

Korrektheit von Partition

```
1   $i := l - 1$ 
2  for  $j := l$  to  $r - 1$  do
3      if  $A[j] \leq A[r]$  then
4           $i := i + 1$ 
5      if  $i < j$  then vertausche
6           $A[i]$  und  $A[j]$ 
```

```
1   $i := l - 1$ ;  $j := l - 1$ 
2  while  $j < r - 1$  do
3       $j := j + 1$ 
4      if  $A[j] \leq A[r]$  then
5           $i := i + 1$ 
6      if  $i < j$  then vertausche
7           $A[i]$  und  $A[j]$ 
```

Beweis der Schleifeninvariante

$A[k] \leq A[r]$ für $k = l, \dots, i$ und $A[k] > A[r]$ für $k = i + 1, \dots, j$.

IS: Zunächst wird j hochgezählt und dann $A[j]$ mit $A[r]$ verglichen.

$A[j] \leq A[r]$: In diesem Fall wird auch i hochgezählt.

Im Fall $i = j$ gilt dann $A[i] = A[j] \leq A[r]$.

Im Fall $i < j$ gilt nach Zeile 5 zunächst $A[i] > A[r]$.

Nach der Vertauschung ist also $A[i] \leq A[r]$ und $A[j] > A[r]$.

QuickSort

Prozedur QuickSort(l, r)

```
1  if  $l < r$  then  $m := \text{Partition}(l, r)$ 
2      QuickSort( $l, m - 1$ )
3      QuickSort( $m + 1, r$ )
```

Laufzeit von QuickSort

- Im **schlechtesten Fall** befindet sich das Pivotelement nach jedem Aufruf von Partition am Rand von A (d.h. $m = l$ oder $m = r$).
- Dieser Fall tritt z.B. ein, wenn das Eingabefeld A bereits sortiert ist.
- Partition wird dann der Reihe nach mit Feldern der Länge $n, n - 1, n - 2, \dots, 2$ aufgerufen.
- Da Partition für ein Feld der Länge n genau $n - 1$ Vergleiche anstellt, benötigt QuickSort im schlechtesten Fall

$$V(n) = \sum_{i=1}^n (i - 1) = \binom{n}{2} = \Theta(n^2)$$

Vergleiche.

QuickSort

Laufzeit von QuickSort

- Im **besten Fall** zerlegt Partition das Feld in 2 gleich große Felder.
- Dann erfüllt $V(n)$ die Rekursion

$$V(n) = \begin{cases} 0, & n = 1, \\ V(\lfloor (n-1)/2 \rfloor) + V(\lceil (n-1)/2 \rceil) + n - 1, & n \geq 2, \end{cases}$$

d.h. $V(n) = n \log_2 n - \Theta(n)$ (siehe worst-case Analyse von MergeSort).

Bemerkung

- Es gibt auch Pivotauswahlstrategien, die in linearer Zeit z.B. den Median bestimmen.
- Dies führt auf eine Variante von QuickSort mit Laufzeit $\Theta(n \log n)$ bei allen Eingaben.
- Allerdings ist die Bestimmung des Medians für praktische Zwecke meist zu aufwändig.

QuickSort

Laufzeit von QuickSort

- Bei der Analyse des **Durchschnittsfalls** gehen wir von einer zufälligen Eingabepermutation $A[1 \dots n]$ der Folge $1, \dots, n$ aus.
- Dann ist die Anzahl $V(n)$ der Vergleichsanfragen von QuickSort eine Zufallsvariable.
- Wir können $V(n)$ als Summe $\sum_{1 \leq i < j \leq n} X_{ij}$ folgender Indikatorvariablen darstellen:

$$X_{ij} = \begin{cases} 1, & \text{falls die Werte } i \text{ und } j \text{ verglichen werden,} \\ 0, & \text{sonst.} \end{cases}$$

- Ob die Werte i und j verglichen werden, entscheidet sich beim ersten Aufruf von $\text{Partition}(l, r)$, bei dem das Pivotelement $x = A[r]$ im Intervall $I_{ij} = \{i, \dots, j\}$ liegt.

QuickSort

Laufzeit von QuickSort

- Ob i und j verglichen werden, entscheidet sich beim ersten Aufruf von $\text{Partition}(l, r)$, bei dem das Pivotelement $x = A[r]$ im Intervall $I_{ij} = \{i, \dots, j\}$ liegt.
- Bis zu diesem Aufruf werden die Werte im Intervall I_{ij} nur mit Pivotelementen außerhalb von I_{ij} verglichen.
- Daher werden sie bis dahin nicht getrennt, sondern bleiben im gleichen Teilfeld beisammen.
- Ist das erste Pivotelement x in I_{ij} nun nicht gleich i oder j , dann werden i und j zwar mit x aber nicht miteinander verglichen.
- Das liegt daran dass im Fall $i < x < j$ die Werte i und j bei diesem Aufruf in zwei verschiedene Teilfelder getrennt werden.
- Die Werte i und j werden also genau dann verglichen, wenn das erste Pivotelement x im Intervall I_{ij} den Wert i oder j hat.

QuickSort

Laufzeit von QuickSort

- Die Werte i und j werden also genau dann verglichen, wenn das erste Pivotelement x im Intervall I_{ij} den Wert i oder j hat.
- Da die Eingabe eine Zufallsfolge ohne Mehrfachvorkommen ist, nimmt x jeden Wert in I_{ij} mit Wahrscheinlichkeit $1/(j-i+1)$ an.
- Folglich findet mit Wahrscheinlichkeit $p_{ij} = 2/(j-i+1)$ ein Vergleich zwischen den Werten i und j statt.
- Der Erwartungswert von $V(n) = \sum_{1 \leq i < j \leq n} X_{ij}$ berechnet sich nun zu

$$\begin{aligned} E[V(n)] &= \sum_{1 \leq i < j \leq n} \underbrace{E[X_{ij}]}_{p_{ij}} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\leq \sum_{i=1}^{n-1} \sum_{k=2}^n \frac{2}{k} \leq 2 \sum_{i=1}^{n-1} \log n = \mathcal{O}(n \log n). \end{aligned}$$

QuickSort

Satz

QuickSort ist ein vergleichendes Sortierverfahren mit einer im Durchschnitt asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.

QuickSort – Zusammenfassung

- QuickSort benötigt im **schlechtesten Fall** $\binom{n}{2}$ Vergleiche und hat daher im schlechtesten Fall eine Laufzeit von $\Theta(n^2)$.
- Im **besten Fall** benötigt QuickSort $n \log_2 n - \Theta(n)$ Vergleiche, was auf eine Laufzeit von $\Theta(n \log n)$ führt.
- Im **Durchschnitt** benötigt QuickSort

$$(2 \ln 2) \log_2(n!) - \Theta(n) = (2 \ln 2) n \log_2 n - \Theta(n)$$

Vergleiche (siehe Übungen), was auf eine durchschnittliche Laufzeit von $\Theta(n \log n)$ führt.

- Ein vergleichendes Sortierverfahren benötigt im Durchschnitt mindestens $\log_2(n!)$ Vergleiche, um eine Folge von n Zahlen zu sortieren (siehe Übungen).
- Die Anzahl der Vergleiche von QuickSort ist also im Durchschnitt nur um den Faktor $2 \ln 2 \approx 1,386$ höher als das Optimum.

Eine randomisierte Variante von QuickSort

- Die Prozedur RandomQuickSort arbeitet ähnlich wie QuickSort.
- Der einzige Unterschied besteht darin, dass als Pivotelement nicht $A[l]$, sondern ein zufälliges Element aus dem Feld $A[l \dots r]$ gewählt wird.

Prozedur RandomQuickSort(l, r)

```
1  if  $l < r$  then
2       $m := \text{RandomPartition}(l, r)$ 
3      RandomQuickSort( $l, m - 1$ )
4      RandomQuickSort( $m + 1, r$ )
```

Prozedur RandomPartition(l, r)

```
1  guess randomly  $j \in \{l, \dots, r\}$ 
2  if  $j < r$  then
3      vertausche  $A[j]$  und  $A[r]$ 
4  return(Partition( $l, r$ ))
```

Eine randomisierte Variante von QuickSort

- Man kann zeigen, dass sich RandomQuickSort bei **jeder** Eingabefolge $A[l, \dots, r]$ exakt gleich verhält wie QuickSort bei einer zufälligen Permutation dieser Eingabefolge (siehe Übungen).
- Daher ist die (erwartete) Laufzeit von RandomQuickSort auch im **schlechtesten Fall** durch $\mathcal{O}(n \log n)$ beschränkt, falls Mehrfachvorkommen ausgeschlossen sind.
- In den Übungen wird gezeigt, dass auch randomisierte vergleichende Sortierverfahren im schlechtesten Fall mindestens $\lceil \log_2(n!) \rceil$ und im Durchschnitt mindestens $\log_2(n!)$ Vergleiche benötigen, um eine Permutation der Zahlen $1, \dots, n$ zu sortieren.

Satz

RandomQuickSort ist ein randomisiertes vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.

HeapSort

Die Prozedur HeapSort benutzt als Datenstruktur einen so genannten **Heap**, um eine Zahlenfolge zu sortieren.

Definition

- Ein **Heap** H mit n Knoten ist ein geordneter Binärbaum der Form



- Das heißt, H hat in Tiefe $i = 0, 1, \dots, \lfloor \log_2 n \rfloor - 1$ jeweils die maximale Anzahl von 2^i Knoten und in Tiefe $\lfloor \log_2 n \rfloor$ sind alle Knoten linksbündig angeordnet.
- Zudem ist jeder Knoten v mit einer Zahl $H[v]$ beschriftet, deren Wert mindestens so groß ist wie die Werte seiner Kinder (sofern vorhanden).

HeapSort

Repräsentation eines Heaps in einem Feld

Ein Heap H lässt sich in einem Feld $H[1, \dots, n]$ speichern. Dabei gilt:

- Das linke Kind von Knoten i hat den Index $\text{left}(i) = 2i$.
- Das rechte Kind von Knoten i hat den Index $\text{right}(i) = 2i + 1$.
- Der Elternknoten von Knoten i hat den Index $\text{parent}(i) = \lfloor i/2 \rfloor$.

Die Heap-Eigenschaft lässt sich nun wie folgt formulieren:

- Für alle Knoten $i \in \{1, \dots, n\}$ gilt:

$$(2i \leq n \Rightarrow H[i] \geq H[2i]) \wedge (2i + 1 \leq n \Rightarrow H[i] \geq H[2i + 1]).$$

- Da die Knoten im Intervall $\{\lfloor n/2 \rfloor + 1, \dots, n\}$ keine Kinder haben, ist für sie die Heap-Eigenschaft automatisch erfüllt.
- Ist $H[1, \dots, n]$ ein Heap, dann repräsentiert auch jedes Anfangsstück $H[1, \dots, r]$, $1 \leq r \leq n$, einen Heap H_r mit r Knoten.
- Zudem ist für $1 \leq i \leq r \leq n$ der Teilbaum von H_r mit Wurzel i ein Heap, den wir mit $H_{i,r}$ bezeichnen.

HeapSort

- Aufgrund der Heap-Eigenschaft muss die Wurzel $H[1]$ eines Heaps den größten Wert haben.
- Daher können wir ein Feld $H[1, \dots, n]$ sortieren, indem wir H zuerst zu einem Heap umsortieren und dann sukzessive
 - die Wurzel $H[1]$ mit dem letzten Feldelement des Heaps vertauschen,
 - den rechten Rand des Heaps um ein Feld nach links verschieben (also die vormalige Wurzel des Heaps herausnehmen) und
 - die durch die Ersetzung der Wurzel verletzte Heap-Eigenschaft wieder herstellen.

HeapSort

- Sei $H[1, \dots, n]$ ein Feld, so dass der Teilbaum $H_{i,r}$ die Heap-Eigenschaft in allen Knoten bis auf seine Wurzel i erfüllt.
- Dann stellt die Prozedur $\text{Heapify}(i, r)$ die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ her.

Prozedur $\text{Heapify}(i, r)$

1	if $(2i \leq r) \wedge (H[2i] > H[i])$ then	linkes Kind ex., größer als Wurzel
2	$x := 2i$	
3	else	
4	$x := i$	rechtes Kind ex., größer als Wurzel
5	if $(2i + 1 \leq r) \wedge (H[2i + 1] > H[x])$ then	
6	$x := 2i + 1$	
7	if $x > i$ then	Kind $>$ Wurzel $\Rightarrow$ vertauschen
8	vertausche $H[x]$ und $H[i]$	
9	$\text{Heapify}(x, r)$	
	Heap-Eigenschaft im Teilbaum herstellen	

HeapSort

Unter Verwendung der Prozedur Heapify ist es nun leicht, ein Feld zu sortieren.

Algorithmus HeapSort

```
1  Input: Ein Feld  $H[1, \dots, n]$ 
2  for  $i := \lfloor n/2 \rfloor$  downto 1 do
3      Heapify( $i, n$ )
4  for  $r := n$  downto 2 do
5      vertausche  $H[1]$  und  $H[r]$ 
6      Heapify( $1, r - 1$ )
```

Beispiel

Wir betrachten die Eingabefolge (6, 3, 1, 7, 9, 2, 4, 8).

HeapSort

Algorithmus HeapSort

```
1 Input: Ein Feld  $H[1, \dots, n]$ 
2 for  $i := \lfloor n/2 \rfloor$  downto 1 do
3   Heapify( $i, n$ )
4 for  $r := n$  downto 2 do
5   vertausche  $H[1]$  und  $H[r]$ 
6   Heapify( $1, r - 1$ )
```

Erste for-Schleife als while-Schleife:

```
1  $i := \lfloor n/2 \rfloor + 1$ 
2 while  $i > 1$  do
3    $i := i - 1$ 
4   Heapify( $i, n$ )
```

Korrektheit von HeapSort

- Unter der Voraussetzung, dass die Prozedur Heapify korrekt arbeitet, folgt die Korrektheit von HeapSort durch den Nachweis folgender Schleifeninvarianten.
- Invariante für die erste for-Schleife:
Für $j = i, \dots, n$ ist der Teilbaum $H_{j,n}$ ein Heap.
- Nach Beendigung dieser Schleife (d.h. $i = 1$) ist also $H_{1,n}$ ein Heap.

HeapSort

Algorithmus HeapSort

```
1 Input: Ein Feld  $H[1, \dots, n]$ 
2 for  $i := \lfloor n/2 \rfloor$  downto 1 do
3   Heapify( $i, n$ )
4 for  $r := n$  downto 2 do
5   vertausche  $H[1]$  und  $H[r]$ 
6   Heapify(1,  $r - 1$ )
```

Zweite for-Schleife als while-Schleife:

```
1  $r := n + 1$ 
2 while  $r > 2$  do
3    $r := r - 1$ 
4   vertausche  $H[1]$  und  $H[r]$ 
5   Heapify(1,  $r - 1$ )
```

Korrektheit von HeapSort

- Invariante für die zweite for-Schleife:

$H[r], \dots, H[n]$ enthalten die $n - r + 1$ größten Feldelemente in sortierter Reihenfolge und $H_{1,r-1}$ ist ein Heap.

- Am Ende der Schleife (d.h. $r = 2$) enthält also $H[2, \dots, n]$ die $n - 1$ größten Elemente in sortierter Reihenfolge, d.h. $H[1, \dots, n]$ ist sortiert.

HeapSort

Prozedur $\text{Heapify}(i, r)$

```
1  if  $(2i \leq r) \wedge (H[2i] > H[i])$  then
2     $x := 2i$ 
3  else
4     $x := i$ 
5  if  $(2i + 1 \leq r) \wedge (H[2i + 1] > H[x])$  then
6     $x := 2i + 1$ 
7  if  $x > i$  then
8    vertausche  $H[x]$  und  $H[i]$ 
9     $\text{Heapify}(x, r)$ 
```

Korrektheit von Heapify

- Sei $H[1, \dots, n]$ ein Feld, so dass der Teilbaum $H_{i,r}$ die Heap-Eigenschaft in allen Knoten bis auf seine Wurzel i erfüllt.
- Dann müssen wir zeigen, dass $\text{Heapify}(i, r)$ die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ herstellt.

Korrektheit von Heapify

Prozedur $\text{Heapify}(i, r)$

```
1  if  $(2i \leq r) \wedge (H[2i] > H[i])$  then  $x := 2i$  else  $x := i$ 
2  if  $(2i + 1 \leq r) \wedge (H[2i + 1] > H[x])$  then  $x := 2i + 1$ 
3  if  $x > i$  then
4    vertausche  $H[x]$  und  $H[i]$ 
5    Heapify( $x, r$ )
```

- $\text{Heapify}(i, r)$ bestimmt den Knoten $x \in \{i, 2i, 2i + 1\}$ mit maximalem Wert $H(x)$.
- Im Fall $x = i$ erfüllt der Knoten i bereits die Heap-Eigenschaft.
- Ist x dagegen eines der Kinder von i , so vertauscht Heapify die Werte von i und x .
- Danach ist die Heap-Eigenschaft höchstens noch im Knoten x verletzt.
- Daher folgt die Korrektheit von Heapify durch einen einfachen Induktionsbeweis über die Rekursionstiefe.

HeapSort

Prozedur Heapify(i, r)

```
1  if  $(2i \leq r) \wedge (H[2i] > H[i])$  then
2     $x := 2i$ 
3  else
4     $x := i$ 
5  if  $(2i + 1 \leq r) \wedge (H[2i + 1] > H[x])$  then
6     $x := 2i + 1$ 
7  if  $x > i$  then
8    vertausche  $H[x]$  und  $H[i]$ 
9    Heapify( $x, r$ )
```

Laufzeit von Heapify

- Es ist leicht zu sehen, dass Heapify(i, r) maximal $2h(i)$ Vergleiche benötigt, wobei $h(i)$ die Höhe des Knotens i in $H_{1,r}$ ist.
- Daher ist die Laufzeit von Heapify(i, r) durch $\mathcal{O}(h(i)) = \mathcal{O}(\log r)$ beschränkt.

HeapSort

Laufzeit von HeapSort

- Für den **Aufbau eines Heaps** H der Tiefe $t = \lfloor \log_2 n \rfloor$ in der ersten for-Schleife wird Heapify für höchstens
 - $2^0 = 1$ Knoten der Höhe $h = t$,
 - $2^1 = 2$ Knoten der Höhe $h = t - 1$,
 - $\vdots$
 - 2^{t-1} Knoten der Höhe $h = t - (t - 1) = 1$ aufgerufen.
- Für $h = 1, \dots, t$ sind das also höchstens 2^{t-h} Knoten der Höhe h .
- Da Heapify für einen Knoten der Höhe h höchstens $2h$ Vergleiche anstellt, benötigt der Aufbau des Heaps maximal

$$V_1(n) = 2 \sum_{h=1}^t h 2^{t-h} \leq 2 \sum_{h=1}^t h \frac{n}{2^h} < 2n \sum_{h=1}^{\infty} \frac{h}{2^h} = 4n$$

Vergleiche.

HeapSort

Algorithmus HeapSort

```
1  Input: Ein Feld  $H[1, \dots, n]$ 
2  for  $i := \lfloor n/2 \rfloor$  downto 1 do
3      Heapify( $i, n$ )
4  for  $r := n$  downto 2 do
5      vertausche  $H[1]$  und  $H[r]$ 
6      Heapify( $1, r - 1$ )
```

Laufzeit von HeapSort

- Für den **Abbau des Heaps** in der zweiten for-Schleife wird Heapify genau $(n - 1)$ -mal aufgerufen.
- Daher benötigt der Abbau des Heaps maximal

$$V_2(n) \leq 2(n - 1)\lfloor \log_2 n \rfloor \leq 2n \log_2 n$$

Vergleiche.

HeapSort

Satz

HeapSort ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $\mathcal{O}(n \log n)$.

HeapSort

Die Floyd-Strategie

- Die **Floyd-Strategie** benötigt zum Abbau des Heaps im Durchschnitt nur halb so viele Vergleiche wie die bisher betrachtete **Williams-Strategie**.
- Die Idee besteht darin, dass $\text{Heapify}(1, r)$ beginnend mit der Wurzel $i_0 = 1$ sukzessive die Werte der beiden Kinder des aktuellen Knotens i_j vergleicht und jeweils zu dem Kind i_{j+1} mit dem größeren Wert absteigt, bis nach $t \leq \lfloor \log_2 r \rfloor$ Schritten ein Blatt i_t erreicht wird.
- Nun geht Heapify auf diesem Pfad bis zum ersten Knoten i_j mit $H[i_j] \geq H[1]$ zurück und führt auf den Werten der Knoten $i_j, i_{j-1}, \dots, i_0$ einen Ringtausch aus, um die Heap-Eigenschaft herzustellen.
- Hierzu werden $t + (t - j + 1) = 2t - j + 1$ Vergleiche benötigt (im Unterschied zu $2j$ Vergleichen bei Williams).
- Da sich der Knoten i_j im Mittel sehr weit unten im Baum befindet, spart man auf diese Art asymptotisch die Hälfte der Vergleiche.

BucketSort

Idee

Sortiere n Zahlen $a_1, \dots, a_n$ aus $[a, b)$ wie folgt (z.B. für $n = 10$, $a = 0$ und $b = 100$):

- 1 Erstelle für $j = 1, \dots, n$ eine Liste L_j für das Teilintervall

$$I_j = \left[a + (j-1) \frac{b-a}{n}, a + j \frac{b-a}{n} \right) = [10(j-1), 10j).$$

- 2 Bestimme zu jedem Element a_i das Intervall I_j , zu dem es gehört, und füge es in die entsprechende Liste L_j ein.
- 3 Sortiere jede Liste L_j .
- 4 Füge die Listen L_j zu einer Liste zusammen.

Einschränkung

Die Schlüssel a_i sollten im Intervall $[a, b)$ möglichst gleichverteilt sein, um Listen L_j von etwa gleicher Länge zu erhalten.

Laufzeit von BucketSort

- Im **schlechtesten** Fall kommen alle Schlüssel in die gleiche Liste.
- Dann hat BucketSort dieselbe asymptotische Laufzeit wie das als Unteroutine verwendete Sortierverfahren.
- Bei annähernd gleichverteilten Schlüsseln ist die **durchschnittliche** Laufzeit von BucketSort $\Theta(n)$.
- Dies gilt sogar, wenn als Unteroutine ein Sortierverfahren der Komplexität $\mathcal{O}(n^2)$ verwendet wird.

BucketSort

Abschätzung der durchschnittlichen Laufzeit von BucketSort

- Wir nehmen an, dass die Folgenglieder a_i im Intervall $[a, b)$ unabhängig gleichverteilt sind.
- Sei X_j die Zufallsvariable, die die Länge von L_j beschreibt.
- Dann ist X_j **binomialverteilt** mit Parametern n und $p = 1/n$.
- Also ist

$$E[X_j] = np = 1 \text{ und } V[X_j] = np(1 - p) = 1 - 1/n < 1.$$

- Wegen $V[X_j] = E[X_j^2] - E[X_j]^2$ ist $E[X_j^2] = V[X_j] + E[X_j]^2 < 2$.
- Daher folgt für die durchschnittliche Laufzeit $T(n)$ von BucketSort:

$$T(n) = \mathcal{O}(n) + E \left[\sum_{j=1}^n \mathcal{O}(X_j^2) \right] = \mathcal{O} \left(n + \sum_{j=1}^n E[X_j^2] \right) = \mathcal{O}(n).$$

CountingSort

Idee

Sortiere ein Feld $A[1, \dots, n]$ mit Einträgen $A[j] \leq k$ durch Zählen der Einträge $A[j] \leq i$, d.h. bestimme $C[i] = \|\{j \mid A[j] \leq i\}\|$ für $i = 0, \dots, k$.

Einschränkung

Die Einträge $A[j]$ sind natürliche Zahlen der Größe höchstens k .

Prozedur CountingSort

```
1  for  $i := 0$  to  $k$  do  $C[i] := 0$ 
2  for  $j := 1$  to  $n$  do  $C[A[j]] := C[A[j]] + 1$ 
3  for  $i := 1$  to  $k$  do  $C[i] := C[i] + C[i - 1]$ 
4  for  $j := 1$  to  $n$  do
5       $B[C[A[j]]] := A[j]$ 
6       $C[A[j]] := C[A[j]] - 1$ 
7  for  $j := 1$  to  $n$  do  $A[j] := B[j]$ 
```

CountingSort

Prozedur CountingSort

```
1  for  $i := 0$  to  $k$  do  $C[i] := 0$ 
2  for  $j := 1$  to  $n$  do  $C[A[j]] := C[A[j]] + 1$ 
3  for  $i := 1$  to  $k$  do  $C[i] := C[i] + C[i - 1]$ 
4  for  $j := 1$  to  $n$  do
5       $B[C[A[j]]] := A[j]$ 
6       $C[A[j]] := C[A[j]] - 1$ 
7  for  $j := 1$  to  $n$  do  $A[j] := B[j]$ 
```

Beispiel

Für $k = 9$ betrachten wir die Eingabefolge $(4, 7, 9, 4, 2, 0, 2)$.

CountingSort

Prozedur CountingSort

```
1  for  $i := 0$  to  $k$  do  $C[i] := 0$ 
2  for  $j := 1$  to  $n$  do  $C[A[j]] := C[A[j]] + 1$ 
3  for  $i := 1$  to  $k$  do  $C[i] := C[i] + C[i - 1]$ 
4  for  $j := 1$  to  $n$  do
5       $B[C[A[j]]] := A[j]$ 
6       $C[A[j]] := C[A[j]] - 1$ 
7  for  $j := 1$  to  $n$  do  $A[j] := B[j]$ 
```

Satz

CountingSort sortiert n natürliche Zahlen der Größe höchstens k in Zeit $\Theta(n + k)$ und Platz $\Theta(n + k)$.

Korollar

CountingSort sortiert n natürliche Zahlen der Größe $\mathcal{O}(n)$ in linearer Zeit und linearem Platz.

RadixSort

Idee

Sortiere d -stellige Zahlen $a = a_d \cdots a_1$ eine Stelle nach der anderen.

Prozedur RadixSort

```
1  for  $i := 1$  to  $d$  do
2    sortiere  $A[1, \dots, n]$  nach der  $i$ -ten Stelle
```

Einschränkung

Die Schlüssel $A[j]$ sollten möglichst als Festkomma-Zahlen vorliegen.

Zudem muss in Zeile 2 „stabil“ sortiert werden.

Definition

Ein Sortierverfahren heißt **stabil**, wenn es die relative Reihenfolge von Elementen mit demselben Wert nicht verändert.

Es empfiehlt sich, eine stabile Variante von CountingSort als Unteroutine zu verwenden.

Eine stabile Variante von CountingSort

Prozedur CountingSort

```
1  for  $i := 0$  to  $k$  do  $C[i] := 0$ 
2  for  $j := 1$  to  $n$  do  $C[A[j]] := C[A[j]] + 1$ 
3  for  $i := 1$  to  $k$  do  $C[i] := C[i] + C[i - 1]$ 
4  for  $j := 1$  to  $n$  do
5       $B[C[A[j]]] := A[j]$ 
6       $C[A[j]] := C[A[j]] - 1$ 
7  for  $j := 1$  to  $n$  do  $A[j] := B[j]$ 
```

RadixSort

Satz

RadixSort sortiert n d -stellige Festkomma-Zahlen zur Basis b in Zeit $\Theta(d(n + b))$.

RadixSort sortiert also n $\mathcal{O}(\log n)$ -stellige Binärzahlen in Zeit $\Theta(n \log n)$. Wenn wir r benachbarte Ziffern zu einer „Ziffer“ $z \in \{0, \dots, b^r - 1\}$ zusammenfassen, erhalten wir folgende Variante von RadixSort.

Korollar

Für jede Zahl $1 \leq r \leq d$ sortiert RadixSort n d -stellige Festkomma-Zahlen zur Basis b in Zeit $\Theta(\frac{d}{r}(n + b^r))$.

Wählen wir beispielsweise $r = \lceil \log_2 n \rceil$, so erhalten wir für $d = \mathcal{O}(\log n)$ -stellige Binärzahlen eine Komplexität von

$$\Theta\left(\frac{d}{r}(n + 2^r)\right) = \Theta(n + 2^r) = \Theta(n).$$

Zusammenfassung

Vergleichsbasierte Sortiervverfahren

	Insertion-Sort	MergeSort	Quick-Sort	Heap-Sort
worst-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
average-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Speicherplatz	$\Theta(1)$	$\Theta(n)$ bzw. $\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
stabil	ja	ja	nein	nein

Linearzeit-Sortierverfahren

- BucketSort: Im Durchschnitt linearer Zeitverbrauch, falls die n Zahlen in einem Intervall $[a, b)$ gleichverteilt sind.
- CountingSort: Sogar im schlechtesten Fall lineare Zeit, falls die Werte natürliche Zahlen sind und $\mathcal{O}(n)$ nicht übersteigen.
- RadixSort: Bitweises Sortieren in linearer Zeit, falls die zu sortierenden Zahlen in Festkomma-Darstellung nicht mehr als $\mathcal{O}(\log n)$ Bit haben.

Datenstrukturen für dynamische Mengen

Dynamische Mengen

- Viele Algorithmen benötigen eine Datenstruktur für dynamische Mengen S .
- Eine solche Datenstruktur sollte im Prinzip **beliebig viele** Elemente aufnehmen können.
- Die Elemente $x \in S$ werden anhand eines **Schlüssels** $k = \text{key}(x)$ identifiziert.
- Auf die Elemente $x \in S$ wird mittels **Zeiger** zugegriffen.

Datenstrukturen für dynamische Mengen

Typische Operationen

Insert(S, x): Fügt x in S ein.

Remove(S, x): Entfernt x aus S .

Search(S, k): Gibt für einen Schlüssel k (einen Zeiger auf) das Element $x \in S$ mit $\text{key}(x) = k$ zurück, falls ein solches Element existiert, und `nil` sonst.

Min(S): Gibt das Element in S mit dem kleinsten Schlüssel zurück.

Max(S): Gibt das Element in S mit dem größten Schlüssel zurück.

Prec(S, x): Gibt das Element in S mit dem nach x nächstkleineren Schlüssel zurück (bzw. `nil`, falls x das Minimum ist).

Succ(S, x): Gibt das Element in S mit dem nach x nächstgrößeren Schlüssel zurück (bzw. `nil`, falls x das Maximum ist).

Verkettete Listen

- Elemente sind in linearer Reihenfolge angeordnet.
- Das erste Element der Liste L ist $\text{head}(L)$.
- Jedes Element x „kennt“ seinen Nachfolger $\text{next}(x)$.

Prozedur $L\text{-Insert}(L, x)$

```
1  next(x) := head(L)
2  head(L) := x
```

- Wenn jedes Element x auch seinen Vorgänger $\text{prev}(x)$ kennt, dann spricht man von einer **doppelt verketteten Liste**.

Prozedur $DL\text{-Insert}(L, x)$

```
1  next(x) := head(L)
2  if head(L)  $\neq$  nil then prev(head(L)) := x
3  head(L) := x
4  prev(x) := nil
```

Doppelt verkettete Listen

Die Prozedur $\text{DL-Remove}(L, x)$ entfernt wieder ein Element x aus einer doppelt verketteten Liste L .

Prozedur $\text{DL-Remove}(L, x)$

```
1  if  $x \neq \text{head}(L)$  then
2     $\text{next}(\text{prev}(x)) := \text{next}(x)$ 
3  else
4     $\text{head}(L) := \text{next}(x)$ 
5  if  $\text{next}(x) \neq \text{nil}$  then
6     $\text{prev}(\text{next}(x)) := \text{prev}(x)$ 
```

Doppelt verkettete Listen

Die Prozedur $\text{DL-Search}(L, k)$ sucht ein Element x mit dem Schlüssel k in der Liste L .

Prozedur $\text{DL-Search}(L, k)$

```
1   $x := \text{head}(L)$   
2  while  $x \neq \text{nil}$  and  $\text{key}(x) \neq k$  do  
3     $x := \text{next}(x)$   
4  return( $x$ )
```

Doppelt verkettete Listen

Laufzeit

- DL-Insert und DL-Remove benötigen konstante Zeit $\Theta(1)$.
- Search hat lineare (in der Länge der Liste) Laufzeit.

Bemerkung

- Würde DL-Remove nur der Schlüssel übergeben, dann wäre die Laufzeit linear, da wir erst mit DL-Search das entsprechende Element suchen müssten.
- Für einfach verkettete Listen ist der Aufwand von Remove linear, da wir keinen Zugriff auf den Vorgänger haben.
- Die Operationen Max, Min, Prec und Succ lassen sich ebenfalls mit linearer Laufzeit implementieren (siehe Übungen).
- Da sich MergeSort für Listen als “in place”-Verfahren implementieren lässt (siehe Übungen), können Listen in konstantem Platz und Zeit $\mathcal{O}(n \log n)$ sortiert werden.

Binärbaum B

- Jedes Element/jeder Knoten x in B hat folgende drei Zeiger:
 - $\text{left}(x)$ zeigt auf das linke Kind,
 - $\text{right}(x)$ zeigt auf das rechte Kind und
 - $\text{parent}(x)$ zeigt auf den Elternknoten.
- Für die Wurzel $w = \text{root}(B)$ ist $\text{parent}(w) = \text{nil}$ und sollte einem Knoten x eines seiner Kinder fehlen, so ist der entsprechende Zeiger ebenfalls nil .
- Auf diese Art lassen sich beispielsweise Heaps für unbeschränkt viele Datensätze implementieren.

Definition

Ein binärer Baum B ist ein **binärer Suchbaum**, falls für jeden Knoten x in B folgende Eigenschaften erfüllt sind:

- Für jeden Knoten y im linken Teilbaum von x gilt

$$\text{key}(y) \leq \text{key}(x)$$

- und für jeden Knoten y im rechten Teilbaum von x gilt

$$\text{key}(y) \geq \text{key}(x).$$

Binäre Suchbäume

Die Prozedur $\text{BST-Search}(B, k)$ sucht ein Element x mit dem Schlüssel k im binären Suchbaum (engl. *search tree*) B .

Prozedur $\text{BST-Search}(B, k)$

```
1   $x := \text{root}(B)$ 
2  while  $x \neq \text{nil} \wedge \text{key}(x) \neq k$  do
3      if  $k \leq \text{key}(x)$  then
4           $x := \text{left}(x)$ 
5      else
6           $x := \text{right}(x)$ 
7  return( $x$ )
```

Binäre Suchbäume

Die Prozedur BST-Insert „sucht“ im Suchbaum B den nil-Zeiger, der auf den neuen Knoten z zeigen müsste, und fügt z an dieser Stelle ein.

Prozedur BST-Insert(B, z)

1	if root(B) = nil then	9	$x := \text{left}(x)$
2	root(B) := z	10	else
3	parent(z) := nil	11	$x := \text{right}(x)$
4	else	12	until $x = \text{nil}$
5	$x := \text{root}(B)$	13	if key(z) $\leq$ key(y) then
6	repeat	14	left(y) := z
7	$y := x$	15	else
8	if key(z) $\leq$ key(x) then	16	right(y) := z
9	$x := \text{left}(x)$	17	parent(z) := y

Satz

Die Operationen BST-Search und BST-Insert laufen auf einem binären Suchbaum der Höhe h in Zeit $\mathcal{O}(h)$.

Bemerkung

Auch die Operationen Min, Max, Succ, Prec und Remove lassen sich auf einem binären Suchbaum der Höhe h in Zeit $\mathcal{O}(h)$ implementieren (siehe Übungen).

Binäre Suchbäume

Laufzeit im schlechtesten Fall

- Die Laufzeiten der Operationen für binäre Suchbäume hängen von der Tiefe der Knoten im Suchbaum ab.
- Suchbäume können zu Listen entarten, z.B. Einfügen von Datensätzen in sortierter Reihenfolge.

⇒ Operationen haben im schlechtesten Fall lineare Laufzeit.

Laufzeit im Durchschnittsfall

- Analyse des Durchschnittsfalls, wenn die Einfügesequenz eine zufällige Permutation von n verschiedenen Zahlen ist.
- Dann lässt sich zeigen, dass der resultierende Suchbaum eine erwartete Tiefe von $\mathcal{O}(\log n)$ hat (siehe Übungen).

⇒ Erwartete Laufzeit der Operationen ist nur $\mathcal{O}(\log n)$.

Balancierte Suchbäume

Idee

- Aktives Ausbalancieren des Suchbaumes während der Einfüge- und Löschooperationen.
- Hierzu gibt es mehrere verschiedene Techniken.
- Die bekanntesten sind
 - Rot-Schwarz-Bäume,
 - Splay-Bäume, und
 - AVL-Bäume.

AVL-Bäume nach Adelson-Velskij und Landis

Definition

- Ein **AVL-Baum** T ist ein binärer Suchbaum, der **höhenbalanciert** ist.
- D.h. für jeden Knoten x von T beträgt die Höhendifferenz zwischen dem linken und rechten Teilbaum von x höchstens eins.
- Dabei setzen wir die Höhe eines nicht existierenden Teilbaumes mit -1 an.

AVL-Bäume haben fast optimale Höhe

Lemma

Die Höhe eines AVL-Baumes mit n Knoten ist $\mathcal{O}(\log n)$.

Beweis.

- Sei $M(h)$ die minimale Blattzahl eines AVL-Baumes der Höhe h .
- Dann gilt

$$M(0) = 1, M(1) = 1, M(2) = 2, M(3) = 3, M(4) = 5, M(5) = 8.$$

- Allgemein gilt

$$M(h) = \begin{cases} 1, & h = 0 \text{ oder } 1, \\ M(h-1) + M(h-2), & h \geq 2. \end{cases}$$

- $M(h)$ ist also die $(h+1)$ -te **Fibonacci-Zahl** F_{h+1} .

AVL-Bäume haben fast optimale Höhe

Beweis.

- $M(h)$ ist also die $(h + 1)$ -te **Fibonacci-Zahl** F_{h+1} .
- Wir zeigen durch Induktion über h , dass $F_{h+1} \geq \phi^{h-1}$ für $h \geq 0$ ist, wobei $\phi = (1 + \sqrt{5})/2$ die "äußere **Goldene Schnittzahl** ist.
- Der Induktionsanfang ($h = 0$ oder 1) ist klar: $F_2 = F_1 = 1 = \phi^0 \geq \phi^{-1}$.
- Unter der IV $F_{h'+1} \geq \phi^{h'-1}$ für $h' \leq h - 1$ folgt wegen $\phi^2 = \phi + 1$

$$F_{h+1} = F_h + F_{h-1} \geq \phi^{h-2} + \phi^{h-3} = \phi^{h-3}(\phi + 1) = \phi^{h-1}.$$

- Daher gilt für die Blattzahl b eines AVL-Baums der Höhe h ,

$$b \geq M(h) = F_{h+1} \geq \phi^{h-1}.$$

- Da ein Binärbaum mit n Knoten höchstens $b \leq (n + 1)/2$ Blätter hat, folgt

$$h \leq 1 + \log_{\phi}(b) < \log_{\phi}(n + 1) = \mathcal{O}(\log_2 n).$$

- Der konstante Faktor in $\mathcal{O}(\log_2 n)$ ist hierbei $\frac{1}{\log_2(\phi)} \approx 1,44$. □

AVL-Bäume als dynamische Datenstruktur

- Für die Aufrechterhaltung der AVL-Eigenschaft eines AVL-Baums T benötigen wir folgende Information über jeden Knoten x .
- Seien h_l und h_r die Höhen des linken und des rechten Teilbaums von x .
- Dann heißt die Höhendifferenz

$$\text{bal}(x) = h_l - h_r$$

die **Balance** von x in T .

- T ist also genau dann ein AVL-Baum, wenn jeder Knoten x in T eine Höhendifferenz zwischen -1 und 1 hat:

$$\forall x : \text{bal}(x) \in \{-1, 0, 1\}.$$

- Im Folgenden bezeichne $T(x)$ den Teilbaum von T mit der Wurzel x .

Einfügen eines Knotens in einen AVL-Baum

Beschreibung der Prozedur AVL-Insert(B, z)

- Wir fügen einen neuen Knoten z in einen AVL-Baum B ähnlich wie die Prozedur BST-Insert für binäre Suchbäume ein.
- D.h. wir „suchen“ den Schlüssel $k = \text{key}(z)$ in B bis wir einen Knoten y mit $k \leq \text{key}(y)$ und $\text{left}(y) = \text{nil}$ bzw. $k > \text{key}(y)$ und $\text{right}(y) = \text{nil}$ erreichen und fügen z an dieser Stelle als Kind von y ein.
- Da z ein Blatt ist, erhält z den Wert $\text{bal}(z) = 0$.
- Das Einfügen von z kann nur für Knoten auf dem Pfad von z zur Wurzel von B eine Änderung der Höhendifferenz bewirken.
- Daher genügt es, diesen Suchpfad zurückzugehen und dabei für jeden besuchten Knoten die AVL-Eigenschaft zu testen und nötigenfalls wiederherzustellen.
- Wir untersuchen zuerst, ob y die AVL-Eigenschaft verletzt.

Einfügen eines Knotens in einen AVL-Baum

Beschreibung der Prozedur AVL-Insert(B, z)

Wir untersuchen zuerst, ob y die AVL-Eigenschaft verletzt.

- 1 Falls der Wert von $\text{bal}(y)$ gleich -1 oder 1 ist, hatte $T(y)$ schon vor dem Einfügen von z die Höhe 1. Daher genügt es, $\text{bal}(y) = 0$ zu setzen.



- 2 Falls $\text{bal}(y) = 0$ ist, wurde z an ein Blatt gehängt, d.h. die Höhe von $T(y)$ ist um 1 gewachsen.

Zunächst setzen wir $\text{bal}(y)$ auf den Wert 1 oder -1 , je nachdem ob z linkes oder rechtes Kind von y ist.



Dann wird die rekursive Prozedur **AVL-Check-Insertion(y)** aufgerufen, die überprüft, ob weitere Korrekturen nötig sind.

Einfügen eines Knotens in einen AVL-Baum

Prozedur AVL-Insert(B, z)

1	if $\text{root}(B) = \text{nil}$ then	14	if $\text{key}(z) \leq \text{key}(y)$ then
2	$\text{root}(B) := z$	15	$\text{left}(y) := z$
3	$\text{parent}(z) := \text{nil}$	16	else
4	$\text{bal}(z) := 0$	17	$\text{right}(y) := z$
5	else	18	$\text{parent}(z) := y$
6	$x := \text{root}(B)$	19	$\text{bal}(z) := 0$
7	repeat	20	if $\text{bal}(y) \in \{-1, 1\}$ then
8	$y := x$	21	$\text{bal}(y) := 0$
9	if $\text{key}(z) \leq \text{key}(x)$ then	22	else
10	$x := \text{left}(x)$	23	if $z = \text{left}(y)$ then
11	else	24	$\text{bal}(y) := 1$
12	$x := \text{right}(x)$	25	else
13	until $(x = \text{nil})$	26	$\text{bal}(y) := -1$
14	if $\text{key}(z) \leq \text{key}(y)$ then	27	AVL-Check-Insertion(B, y)

Einfügen eines Knotens in einen AVL-Baum

Beschreibung der Prozedur AVL-Check-Insertion(B, y)

Wir setzen voraus, dass bei jedem Aufruf der Prozedur AVL-Check-Insertion(B, y) folgende Invariante gilt:

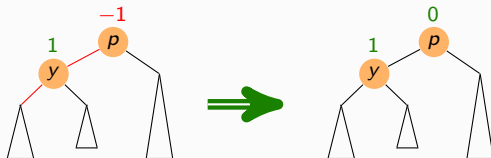
Invariante für AVL-Check-Insertion(B, y)

Der Wert von $\text{bal}(y)$ wurde von 0 auf ± 1 aktualisiert, d.h. die Höhe von $T(y)$ ist um 1 gewachsen.

Falls y die Wurzel von B ist, ist nichts weiter zu tun.

Andernfalls nehmen wir an, dass y linkes Kind von $p = \text{parent}(y)$ ist (der Fall $y = \text{right}(p)$ ist analog).

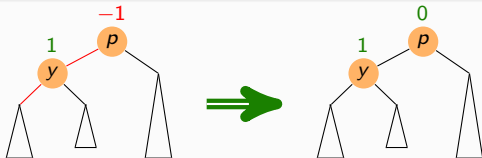
- 1 Im Fall $\text{bal}(p) = -1$ genügt es, $\text{bal}(p) = 0$ zu setzen.



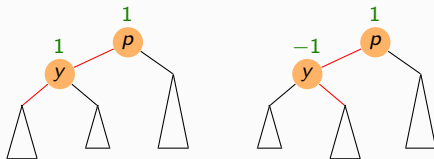
Einfügen eines Knotens in einen AVL-Baum

Beschreibung der Prozedur AVL-Check-Insertion(B, y)

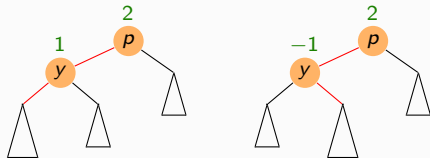
- ❶ Im Fall $\text{bal}(p) = -1$ genügt es, $\text{bal}(p) = 0$ zu setzen.



- ❷ Im Fall $\text{bal}(p) = 0$ setzen wir $\text{bal}(p) = 1$ und rufen AVL-Check-Insertion(p) auf.



- ❸ Im Fall $\text{bal}(p) = 1$ müssen wir B umstrukturieren, da die aktuelle Höhendifferenz von p gleich 2 ist.

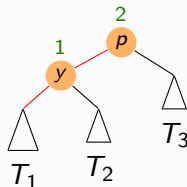


Einfügen eines Knotens in einen AVL-Baum

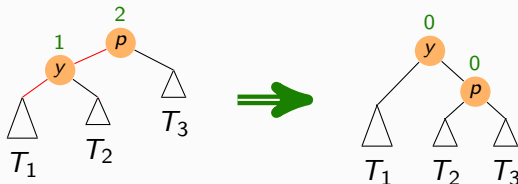
Beschreibung der Prozedur AVL-Check-Insertion(B, y)

- 3a Im Fall $\text{bal}(y) = 1$ sei T_1 der linke und T_2 der rechte Teilbaum von y . Weiter sei T_3 der rechte Teilbaum von p und h sei die Höhe von $T(y)$. Dann gilt für die Höhen h_i der Teilbäume T_i :

$$h_1 = h - 1 \text{ und } h_2 = h_3 = h - 2.$$



Falls wir also eine **Rechtsrotation** ausführen,



hat der rotierte Teilbaum wieder die gleiche Höhe wie vor dem Einfügen von z . Daher ist nichts weiter zu tun.

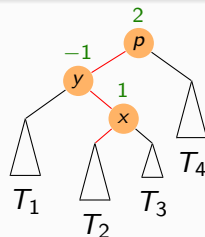
Einfügen eines Knotens in einen AVL-Baum

Beschreibung der Prozedur AVL-Check-Insertion(B, y)

- 3b Im Fall $\text{bal}(y) = -1$ sei T_1 der linke Teilbaum von y und T_4 der rechte Teilbaum von p . Weiter seien T_2 und T_3 der linke und rechte Teilbaum von $x = \text{right}(y)$. Die Höhe von $T(y)$ bezeichnen wir wieder mit h . Dann gilt

$$h_1 = h_4 = h - 2 \text{ und } h - 3 \leq h_2, h_3 \leq h - 2,$$

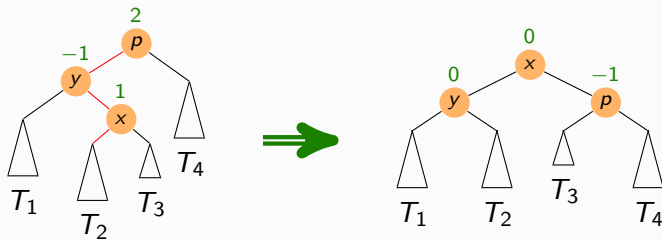
wobei $h_3 = h_2 - \text{bal}(x)$ ist.



Einfügen eines Knotens in einen AVL-Baum

Beschreibung der Prozedur AVL-Check-Insertion(B, y)

- 3b) Daher genügt es, eine **Doppelrotation** (genauer: eine **Links-Rechts-Rotation**) auszuführen:



Die neuen Balance-Werte sind

$$\text{bal}(y) = \begin{cases} 1, & \text{bal}(x) = -1, \\ 0, & \text{sonst,} \end{cases} \quad \text{bal}(p) = \begin{cases} -1, & \text{bal}(x) = 1, \\ 0, & \text{sonst.} \end{cases}$$

und $\text{bal}(x) = 0$.

Der rotierte Teilbaum hat die gleiche Höhe wie der ursprüngliche Teilbaum an dieser Stelle und daher ist nichts weiter zu tun.

AVL-Check-Insertion

Prozedur AVL-Check-Insertion(B, y)

```
1   $p := \text{parent}(y)$ 
2  if  $p = \text{nil}$  then return
3  if  $y = \text{left}(p)$  then
4    if  $\text{bal}(p) = -1$  then
5       $\text{bal}(p) := 0$ 
6    else if  $\text{bal}(p) = 0$  then
7       $\text{bal}(p) := 1$ 
8      AVL-Check-Insertion( $B, p$ )
9    else //  $\text{bal}(p) = 1$ 
10     if  $\text{bal}(y) = 1$  then
11       RightRotate( $B, y$ )
12     else //  $\text{bal}(y) = -1$ 
13       LeftRightRotate( $B, y$ )
14  else //  $y = \text{right}(p)$ 
15    if  $\text{bal}(p) = 1$  then
16       $\text{bal}(p) := 0$ 
17    else if  $\text{bal}(p) = 0$  then
18       $\text{bal}(p) := -1$ 
19      AVL-Check-Insertion( $B, p$ )
20    else //  $\text{bal}(p) = -1$ 
21     if  $\text{bal}(y) = -1$  then
22       LeftRotate( $B, y$ )
23     else //  $\text{bal}(y) = 1$ 
24       RightLeftRotate( $B, y$ )
```

Die Rechtsrotation

Prozedur $\text{RightRotate}(B, y)$

```
1   $p := \text{parent}(y)$ 
2   $T_1 := \text{left}(y)$ 
3   $T_2 := \text{right}(y)$ 
4   $T_3 := \text{right}(p)$ 
5  // setze  $y$  an die Wurzel
   des Teilbaums  $T(p)$ 
6   $p' := \text{parent}(p)$ 
7  if  $p' = \text{nil}$  then
8     $\text{root}(B) := y$ 
9  else if  $\text{left}(p') = p$  then
10    $\text{left}(p') := y$ 
11  else //  $\text{right}(p') = p$ 
12    $\text{right}(p') := y$ 
13   $\text{parent}(y) := p'$ 
14  // setze  $p$  als rechtes
   Kind von  $y$ 
15   $\text{right}(y) := p$ 
16   $\text{parent}(p) := y$ 
17  // setze  $T_2$  als linken
   Teilbaum von  $p$ 
18   $\text{left}(p) := T_2$ 
19  if  $T_2 \neq \text{nil}$  then
20    $\text{parent}(T_2) := p$ 
21  // aktualisiere die
   Balancen
22   $\text{bal}(y) := 0$ 
    $\text{bal}(p) := 0$ 
```

Zusammenfassung

Dynamische Mengen (worst-case Komplexitäten)

	Search	Min/Max	Prec/Succ	Insert	Remove
Heap	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Liste (einfach)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Liste (doppelt)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Suchbaum	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
AVL-Baum	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Notation – ungerichtete Graphen

Definition

- Ein (ungerichteter) Graph ist ein Paar $G = (V, E)$, wobei
 - V - eine endliche Menge von Knoten/Ecken und
 - E - die Menge der Kanten ist.

Hierbei gilt

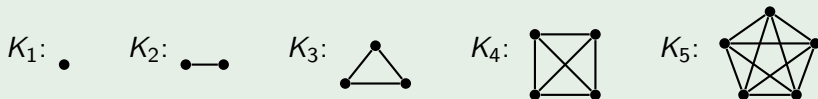
$$E \subseteq \binom{V}{2} := \{\{u, v\} \subseteq V \mid u \neq v\}.$$

- Die Nachbarschaft von v ist $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$.
- Der Grad von v ist $\deg_G(v) = \|N_G(v)\|$.
- Der Minimalgrad von G ist $\delta(G) := \min_{v \in V} \deg_G(v)$ und
- der Maximalgrad von G ist $\Delta(G) := \max_{v \in V} \deg_G(v)$.

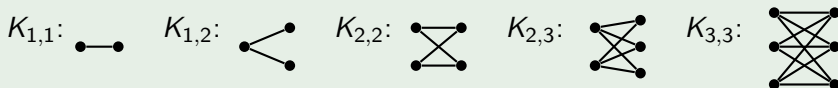
Falls G aus dem Kontext ersichtlich ist, schreiben wir auch einfach $N(v)$, $\deg(v)$, δ usw.

Beispiel

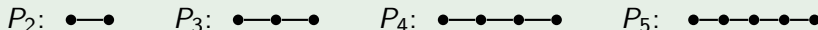
- Der **vollständige Graph** (V, E) mit $\|V\| = n$ und $E = \binom{V}{2}$ wird mit K_n und der **leere Graph** $(V, \emptyset)$ wird mit E_n bezeichnet.



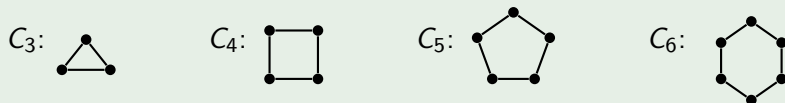
- Der **vollständige bipartite Graph** (A, B, E) auf $a + b$ Knoten, d.h. $A \cap B = \emptyset$, $\|A\| = a$, $\|B\| = b$ und $E = \{\{u, v\} \mid u \in A, v \in B\}$ wird mit $K_{a,b}$ bezeichnet.



- Der **Pfad der Länge $n - 1$** wird mit P_n bezeichnet.



- Der **Kreis der Länge n** wird mit C_n bezeichnet.



Subgraphen, Wege und Pfade

Definition

- Ein Graph $H = (V', E')$ heißt **Sub-/Teil-/Untergraph** von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$ ist.

Der Subgraph H heißt (**durch V'**) **induziert**, falls $E' = E \cap \binom{V'}{2}$ ist. Hierfür schreiben wir auch $H = G[V']$.

- Ein **Weg** ist eine Folge von Knoten $v_0, \dots, v_j$ mit $\{v_i, v_{i+1}\} \in E$ für $i = 0, \dots, j-1$.

Die **Länge** des Weges ist die Anzahl der Kanten, also j .

Ein Weg $v_0, \dots, v_j$ heißt auch **v_0 - v_j -Weg**.

- Ein Weg heißt **einfach** oder **Pfad**, falls alle durchlaufenen Knoten verschieden sind.

Zyklen, Kreise und Bäume

Definition

- Ein **Zyklus** ist ein u - v -Weg der Länge $j \geq 2$ mit $u = v$.
- Ein **Kreis** ist ein Zyklus $v_0, v_1, \dots, v_{j-1}, v_0$ der Länge $j \geq 3$, für den $v_0, v_1, \dots, v_{j-1}$ paarweise verschieden sind.
- Ein Graph $G = (V, E)$ heißt **kreisfrei**, **azyklisch** oder **Wald**, falls er keinen Kreis enthält. Ein **Baum** ist ein zusammenhängender Wald.
- Jeder Knoten $u \in V$ vom Grad $\deg(u) \leq 1$ heißt **Blatt** und die übrigen Knoten (vom Grad ≥ 2) heißen **innere Knoten**.

Notation – gerichtete Graphen

Definition

- Ein **gerichteter Graph** oder **Digraph** ist ein Paar $G = (V, E)$, wobei
 - V - eine endliche Menge von **Knoten/Ecken** und
 - E - die Menge der **Kanten** ist.

Hierbei gilt

$$E \subseteq V \times V = \{(u, v) \mid u, v \in V\},$$

wobei E auch Schlingen (u, u) enthalten kann.

- Die **Nachfolgermenge** von v ist $N^+(v) = \{u \in V \mid (v, u) \in E\}$.
- Die **Vorgängermenge** von v ist $N^-(v) = \{u \in V \mid (u, v) \in E\}$.
- Die **Nachbarmenge** von v ist $N(v) = N^+(v) \cup N^-(v)$.
- Der **Ausgangsgrad** von v ist $\deg^+(v) = \|N^+(v)\|$.
- Der **Eingangsgrad** von v ist $\deg^-(v) = \|N^-(v)\|$.
- Der **Grad** von v ist $\deg(v) = \deg^+(v) + \deg^-(v)$.

Gerichtete Zyklen und Kreise

Definition

Sei $G = (V, E)$ ein Digraph.

- Ein (gerichteter) v_0 - v_j -Weg in G ist eine Folge von Knoten $v_0, \dots, v_j$ mit $(v_i, v_{i+1}) \in E$ für $i = 0, \dots, j-1$.
- Ein (gerichteter) Zyklus in G ist ein gerichteter u - v -Weg der Länge $j \geq 1$ mit $u = v$.
- Ein gerichteter Weg heißt einfach oder (gerichteter) Pfad, falls alle durchlaufenen Knoten paarweise verschieden sind.
- Ein (gerichteter) Kreis in G ist ein gerichteter Zyklus $v_0, v_1, \dots, v_{j-1}, v_0$ der Länge $j \geq 1$, für den $v_0, v_1, \dots, v_{j-1}$ paarweise verschieden sind.
- G heißt azyklisch, wenn es in G keinen gerichteten Kreis gibt.

Datenstrukturen für Graphen

Adjazenzmatrix

- Sei $G = (V, E)$ ein ungerichteter bzw. gerichteter Graph.
- Sei $V = \{v_1, \dots, v_n\}$.
- Dann ist die $(n \times n)$ -Matrix $A = (a_{ij})$ mit Einträgen

$$a_{ij} = \begin{cases} 1, & \{v_i, v_j\} \in E \\ 0, & \text{sonst} \end{cases} \quad \text{bzw.} \quad a_{ij} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & \text{sonst} \end{cases}$$

die **Adjazenzmatrix** von G .

- Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch mit $a_{ij} = 0$ für $i = 1, \dots, n$.

Datenstrukturen für Graphen

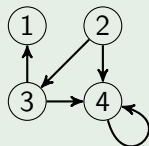
Adjazenzlisten-Darstellung

- Für jeden Knoten v_i wird eine Liste mit seinen Nachbarn verwaltet.
- Im gerichteten Fall verwaltet man entweder nur die Liste der Nachfolger oder zusätzlich eine weitere für die Vorgänger.
- Falls die Anzahl der Knoten konstant bleibt, organisiert man die Adjazenzlisten in einem Feld, d.h. das Feldelement mit Index i verweist auf die Adjazenzliste von Knoten v_i .
- Falls sich die Anzahl der Knoten dynamisch ändert, werden die Adjazenzlisten typischerweise ebenfalls in einer doppelt verketteten Liste verwaltet.

Darstellung von endlichen Graphen

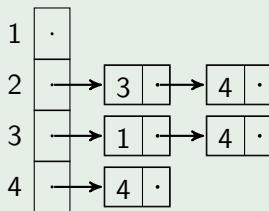
Beispiel

- Betrachte den gerichteten Graphen $G = (V, E)$ mit $V = \{1, 2, 3, 4\}$ und $E = \{(2, 3), (2, 4), (3, 1), (3, 4), (4, 4)\}$.



- Dieser hat folgende Adjazenzmatrix- und Adjazenzlisten-Darstellung:

	1	2	3	4
1	0	0	0	0
2	0	0	1	1
3	1	0	0	1
4	0	0	0	1



Datenstrukturen für Graphen – Aufwand

	Adjazenzmatrix		Adjazenzlisten	
	einfach	clever	einfach	clever
Speicherbedarf	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n + m)$	$\mathcal{O}(n + m)$
Initialisieren	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante entfernen	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Test auf Kante	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

- Der Aufwand für die Initialisierung des leeren Graphen in der Adjazenzmatrix- und Adjazenzlistendarstellung lässt sich auf $\mathcal{O}(1)$ drücken, indem man mithilfe eines zusätzlichen Feldes B die Gültigkeit der Feldeinträge verwaltet (siehe Übungen).
- Die Verbesserung beim Löschen einer Kante in der Adjazenzlistendarstellung erhält man, indem man die Adjazenzlisten doppelt verkettet und im ungerichteten Fall die beiden Vorkommen jeder Kante in den Adjazenzlisten der beiden Endknoten gegenseitig verlinkt.

Elementare Digraphoperationen (Feld von Adjazenzlisten)

Prozedur Init(G)

```
1  erzeuge Feld  $a[1, \dots, n]$ 
2  for  $i := 1$  to  $n$  do
3     $a[i] := \text{nil}$ 
```

Prozedur DiInsertEdge(u, v)

```
1  erzeuge Listeneintrag  $e$ 
2   $\text{source}(e) := u$ 
3   $\text{target}(e) := v$ 
4   $\text{prev}(e) := \text{nil}$ 
5   $\text{next}(e) := a[u]$ 
6  if  $a[u] \neq \text{nil}$  then
7     $\text{prev}(a[u]) := e$ 
8   $a[u] := e$ 
9  return  $e$ 
```

Prozedur DiRemoveEdge(e)

```
1  if  $\text{next}(e) \neq \text{nil}$  then
2     $\text{prev}(\text{next}(e)) := \text{prev}(e)$ 
3  if  $\text{prev}(e) \neq \text{nil}$  then
4     $\text{next}(\text{prev}(e)) := \text{next}(e)$ 
5  else
6     $a[\text{source}(e)] := \text{next}(e)$ 
```

Prozedur Edge(u, v)

```
1   $e := a[u]$ 
2  while  $e \neq \text{nil}$  do
3    if  $\text{target}(e) = v$  then
4      return 1
5     $e := \text{next}(e)$ 
6  return 0
```

Elementare Graphoperationen (Feld von Adjazenzlisten)

Prozedur Init(G)

```
1  erzeuge Feld  $a[1, \dots, n]$ 
2  for  $i := 1$  to  $n$  do
3     $a[i] := \text{nil}$ 
```

Prozedur RemoveEdge(e)

```
1  DiRemoveEdge( $e$ )
2  DiRemoveEdge(opposite( $e$ ))
```

Prozedur Edge(u, v)

```
1   $e := a[u]$ 
2  while  $e \neq \text{nil}$  do
3    if target( $e$ ) =  $v$  then
4      return 1
5     $e := \text{next}(e)$ 
6  return 0
```

Prozedur InsertEdge(u, v)

```
1  erzeuge Listeneinträge  $e, e'$ 
2  opposite( $e$ ) :=  $e'$ 
3  opposite( $e'$ ) :=  $e$ 
4  next( $e$ ) :=  $a[u]$ 
5  next( $e'$ ) :=  $a[v]$ 
6  if  $a[u] \neq \text{nil}$  then
7    prev( $a[u]$ ) :=  $e$ 
8  if  $a[v] \neq \text{nil}$  then
9    prev( $a[v]$ ) :=  $e'$ 
10  $a[u] := e$ ;  $a[v] := e'$ 
11 source( $e$ ) := target( $e'$ ) :=  $u$ 
12 target( $e$ ) := source( $e'$ ) :=  $v$ 
13 prev( $e$ ) := nil
14 prev( $e'$ ) := nil
15 return  $e$ 
```

Durchsuchen von Graphen

- Wir starten die Suche in einem beliebigen Knoten u .
- Für die Suche ist es vorteilhaft, die bereits besuchten (aber noch nicht abgearbeiteten) Knoten in einer Menge B zu speichern.
- Damit die Suche effizient ist, sollte die Datenstruktur für B folgende Operationen effizient implementieren.

Benötigte Operationen

$\text{Init}(B)$: Initialisiert B als leere Menge.

$\text{Empty}(B)$: Testet B auf Leerheit.

$\text{Insert}(B, u)$: Fügt u in B ein.

$\text{Element}(B)$: Gibt ein Element aus B zurück.

$\text{Remove}(B)$: Gibt ebenfalls $\text{Element}(B)$ zurück und entfernt es aus B .

Andere Operationen wie z.B. $\text{Remove}(B, u)$ werden nicht benötigt.

Keller (engl. stack) und Warteschlange (queue)

- Die gewünschten Operationen lassen sich leicht durch einen **Keller** (auch **Stapel** genannt) oder eine **Warteschlange** implementieren.
- Falls die Größe von B maximal n ist, kann ein Feld zur Speicherung der Elemente benutzt werden.
- Andernfalls kann auch eine einfach verkettete Liste benutzt werden.

Stack S – Last-In-First-Out

$\text{Top}(S)$: Gibt das oberste Element von S zurück.

$\text{Push}(S, x)$: Fügt x als oberstes Element zum Keller S hinzu.

$\text{Pop}(S)$: Gibt das oberste Element von S zurück und entfernt es.

Queue Q – Last-In-Last-Out

$\text{Head}(Q)$: Gibt das erste Element von Q zurück.

$\text{Enqueue}(Q, x)$: Fügt x als letztes Element der Schlange Q hinzu.

$\text{Dequeue}(Q)$: Gibt das erste Element von Q zurück und entfernt es.

Implementation eines Kellers als Feld $S[1 \dots n]$

Die Variable $\text{size}(S)$ enthält die Anzahl der im Keller gespeicherten Elemente.

StackInit(S)

```
1  size( $S$ ) := 0
```

StackEmpty(S)

```
1  return(size( $S$ ) = 0)
```

Push(S, x)

```
1  if size( $S$ ) <  $n$  then
2    size( $S$ ) := size( $S$ ) + 1
3     $S[\text{size}(\mathbf{S})]$  :=  $x$ 
4  else
5    return(nil)
```

Top(S)

```
1  if size( $S$ ) > 0 then
2    return( $S[\text{size}(\mathbf{S})]$ )
3  else
4    return(nil)
```

Pop(S)

```
1  if size( $S$ ) > 0 then
2    size( $S$ ) := size( $S$ ) - 1
3    return( $S[\text{size}(\mathbf{S}) + 1]$ )
4  else
5    return(nil)
```

Implementation einer Warteschlange als Feld $Q[1 \dots n]$

$\text{head}(Q)$ ist Index des ersten, $\text{tail}(Q) - 1$ der des letzten Elements in S .

Head(Q)

```
1  if size(Q) = 0 then
2    return(nil)
3  else
4    return Q[head(Q)]
```

QueueInit(Q)

```
1  head(Q) := 1
2  tail(Q) := 1
3  size(Q) := 0
```

Enqueue(Q, x)

```
1  if size(Q) =  $n$  then
2    return(nil)
3  size(Q) := size(Q) + 1
4  Q[tail(Q)] :=  $x$ 
5  if tail(Q) =  $n$  then
6    tail(Q) := 1
7  else
8    tail(Q) := tail(Q) + 1
```

Dequeue(Q)

```
1  if size(Q) = 0 then
2    return(nil)
3  size(Q) := size(Q) - 1
4   $x$  := Q[head(Q)]
5  if head(Q) =  $n$  then
6    head(Q) := 1
7  else
8    head(Q) := head(Q) + 1
9  return( $x$ )
```

Komplexität der Operationen

Satz

Sämtliche Operationen für einen Keller S und eine Warteschlange Q sind in konstanter Zeit $\mathcal{O}(1)$ ausführbar.

Bemerkung

Mit Hilfe von einfach verketteten Listen sind Keller und Warteschlangen auch für eine unbeschränkte Anzahl von Datensätzen mit denselben Laufzeitbeschränkungen implementierbar.

Durchsuchen von Graphen

Realisierung der Datenstruktur B mittels Keller bzw. Schlange

Operation	Keller S	Schlange Q
Init(B)	StackInit(S)	QueueInit(Q)
Empty(B)	StackEmpty(S)	QueueEmpty(Q)
Insert(B, u)	Push(S, u)	Enqueue(Q, u)
Element(B)	Top(S)	Head(Q)
Remove(B)	Pop(S)	Dequeue(Q)

Durchsuchen von Graphen

Wir geben nun für die Suche in einem Graphen bzw. Digraphen $G = (V, E)$ einen Algorithmus `GraphSearch` mit folgenden Eigenschaften an:

- `GraphSearch` benutzt eine Prozedur `Explore`, um alle Knoten und Kanten von G zu besuchen.
- `Explore(w)` findet Pfade zu allen von w aus erreichbaren Knoten.
- Hierzu speichert `Explore(w)` für jeden über eine Kante $\{u, v\}$ bzw. (u, v) neu entdeckten Knoten $v \neq w$ den Knoten u in `parent(v)`.

Algorithmus `GraphSearch(V, E)`

```
1  for all  $v \in V, e \in E$  do
2       $\text{vis}(v) := \text{false}; \text{parent}(v) := \text{nil}; \text{vis}(e) := \text{false}$ 
3  for all  $w \in V$  do
4      if  $\text{vis}(w) = \text{false}$  then Explore(w)
```

Durchsuchen von Graphen

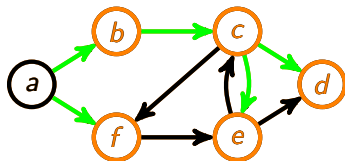
Algorithmus $\text{GraphSearch}(V, E)$

```
1  for all  $v \in V, e \in E$  do
2     $\text{vis}(v) := \text{false}; \text{parent}(v) := \text{nil}; \text{vis}(e) := \text{false}$ 
3  for all  $w \in V$  do
4    if  $\text{vis}(w) = \text{false}$  then  $\text{Explore}(w)$ 
```

Prozedur $\text{Explore}(w)$

```
1   $\text{vis}(w) := \text{true}; \text{Init}(B); \text{Insert}(B, w)$ 
2  while  $\neg \text{Empty}(B)$  do
3     $u := \text{Element}(B)$ 
4    if  $\exists e = \{u, v\}$  bzw.  $e = (u, v) \in E : \text{vis}(e) = \text{false}$  then
5       $\text{vis}(e) := \text{true}$ 
6      if  $\text{vis}(v) = \text{false}$  then
7         $\text{vis}(v) := \text{true}; \text{parent}(v) := u; \text{Insert}(B, v)$ 
8    else  $\text{Remove}(B)$ 
```

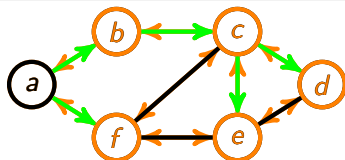
Durchsuchen eines Digraphen



Menge B	bes. Knoten	bes. Kante	neu
$\{a\}$	a	(a, b)	b
$\{a, b\}$	a	(a, f)	f
$\{a, b, f\}$	a	-	-
$\{b, f\}$	b	(b, c)	c
$\{b, c, f\}$	b	-	-
$\{c, f\}$	c	(c, d)	d
$\{c, d, f\}$	c	(c, e)	e
$\{c, d, e, f\}$	c	(c, f)	-

B	bes. Knoten	bes. Kante	neu
$\{c, d, e, f\}$	c	-	-
$\{d, e, f\}$	d	-	-
$\{e, f\}$	e	(e, c)	-
$\{e, f\}$	e	(e, d)	-
$\{e, f\}$	e	-	-
$\{f\}$	f	(f, e)	-
$\{f\}$	f	-	-
$\emptyset$			

Durchsuchen eines Graphen



Menge B	Knoten	Kante	neu	B	Knoten	Kante	neu
$\{a\}$	a	$\{a, b\}$	b	$\{d, e, f\}$	d	$\{d, c\}$	-
$\{a, b\}$	a	$\{a, f\}$	f	$\{d, e, f\}$	d	$\{d, e\}$	-
$\{a, b, f\}$	a	-	-	$\{d, e, f\}$	d	-	-
$\{b, f\}$	b	$\{b, a\}$	-	$\{e, f\}$	e	$\{e, c\}$	-
$\{b, f\}$	b	$\{b, c\}$	c	$\{e, f\}$	e	$\{e, d\}$	-
$\{b, c, f\}$	b	-	-	$\{e, f\}$	e	$\{e, f\}$	-
$\{c, f\}$	c	$\{c, b\}$	-	$\{e, f\}$	e	-	-
$\{c, d, f\}$	c	$\{c, d\}$	d	$\{f\}$	f	$\{f, a\}$	-
$\{c, d, f\}$	c	$\{c, e\}$	e	$\{f\}$	f	$\{f, c\}$	-
$\{c, d, e, f\}$	c	$\{c, f\}$	-	$\{f\}$	f	$\{f, e\}$	-
$\{c, d, e, f\}$	c	-	-	$\{f\}$	f	-	-

Gerichtete Bäume und Wälder

Definition

- Ein Digraph G heißt **gerichteter Wald**, wenn G azyklisch ist und jeder Knoten $v \in V$ Eingangsgrad $\deg^-(v) \leq 1$ hat.
- Ein Knoten $u \in V$ vom Ausgangsgrad $\deg^+(u) = 0$ heißt **Blatt**.
- Ein Knoten $u \in V$ heißt **Nachfahre** von v , falls in G ein gerichteter v - u -Weg existiert. In diesem Fall ist v ein **Vorfahre** von u . Gilt zudem $u \neq v$, so sprechen wir auch von **echten** Nach- bzw. Vorfahren.
- Ein Knoten $w \in V$ heißt **Wurzel** von G , falls alle Knoten $v \in V$ von w aus erreichbar sind (d.h. es gibt einen gerichteten w - v -Weg in G).
- Ein **gerichteter Wald**, der eine Wurzel hat, heißt **gerichteter Baum**.
- In einem gerichteten Baum liegen die Kantenrichtungen durch die Wahl der Wurzel bereits eindeutig fest.
- Daher kann man bei bekannter Wurzel auf die Angabe der Kantenrichtungen verzichten.
- Man spricht dann auch von einem **Wurzelbaum**.

Suchwälder

Bemerkung

- Wir bezeichnen den durch $\text{SearchGraph}(V, E)$ erzeugten Digraphen $W = (V, E_{\text{parent}})$ mit

$$E_{\text{parent}} = \{(\text{parent}(v), v) \mid v \in V \text{ und } \text{parent}(v) \neq \text{nil}\}$$

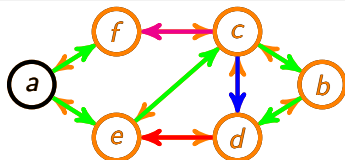
als **Suchwald** von G .

- Die Kanten $(\text{parent}(v), v)$ von W heißen **Baumkanten**.
- Da $\text{parent}(v)$ vor v besucht wird, ist klar, dass W kreisfrei ist.
- Zudem hat jeder Knoten v höchstens einen Vorgänger $\text{parent}(v)$.
- Dies zeigt, dass W tatsächlich ein gerichteter Wald ist.
- W hängt zum einen davon ab, wie die Datenstruktur B implementiert ist (z.B. als Keller oder als Warteschlange).
- Zum anderen hängt W aber auch von der Reihenfolge der Knoten in den Adjazenzlisten ab.

Klassifikation der Kanten eines (Di-)Graphen

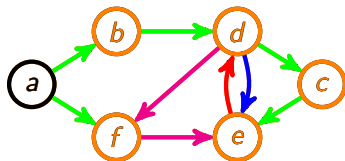
- Die Kanten eines Graphen $G = (V, E)$ werden durch den Suchwald $W = (V, E_{\text{parent}})$ in vier Klassen eingeteilt.
- Dabei erhält jede Kante die Richtung, in der sie bei ihrem ersten Besuch durchlaufen wird.
- Neben den Baumkanten $(\text{parent}(v), v) \in E_{\text{parent}}$ gibt es noch Rückwärts-, Vorwärts- und Querkanten.
- Rückwärtskanten (u, v) verbinden einen Knoten u mit einem Knoten v , der auf dem parent-Pfad $P(u)$ von u liegt.
- Liegt dagegen u auf $P(v)$, so wird (u, v) als Vorwärtskante bezeichnet.
- Alle übrigen Kanten heißen Querkanten. Diese verbinden zwei Knoten, von denen keiner auf dem parent-Pfad des anderen liegt.
- Die Klassifikation der Kanten eines Digraphen G erfolgt analog, wobei die Richtungen jedoch bereits durch G vorgegeben sind (dabei werden Schlingen der Kategorie der Vorwärtskanten zugeordnet).

Klassifikation der Kanten eines Graphen



Menge B	Knoten	Kante	Typ	B	Knoten	Kante	Typ
$\{a\}$	a	$\{a, e\}$	B	$\{c, d, e, f\}$	c	$\{c, f\}$	Q
$\{a, e\}$	a	$\{a, f\}$	B	$\{c, d, e, f\}$	c	-	-
$\{a, e, f\}$	a	-	-	$\{d, e, f\}$	d	$\{d, b\}$	-
$\{e, f\}$	e	$\{e, a\}$	-	$\{d, e, f\}$	d	$\{d, c\}$	-
$\{e, f\}$	e	$\{e, c\}$	B	$\{d, e, f\}$	d	$\{d, e\}$	R
$\{c, e, f\}$	c	$\{c, b\}$	B	$\{d, e, f\}$	d	-	-
$\{b, c, e, f\}$	b	$\{b, c\}$	-	$\{e, f\}$	e	$\{e, d\}$	-
$\{b, c, e, f\}$	b	$\{b, d\}$	B	$\{e, f\}$	e	-	-
$\{b, c, d, e, f\}$	b	-	-	$\{f\}$	f	$\{f, a\}$	-
$\{c, d, e, f\}$	c	$\{c, d\}$	V	$\{f\}$	f	$\{f, c\}$	-
$\{c, d, e, f\}$	c	$\{c, e\}$	-	$\{f\}$	f	-	-

Klassifikation der Kanten eines Digraphen



Menge B	bes. Knoten	bes. Kante	Typ
$\{a\}$	a	(a, b)	B
$\{a, b\}$	a	(a, f)	B
$\{a, b, f\}$	a	-	-
$\{b, f\}$	b	(b, d)	B
$\{b, d, f\}$	b	-	-
$\{d, f\}$	d	(d, c)	B
$\{c, d, f\}$	c	(c, e)	B
$\{c, d, e, f\}$	c	-	-

B	bes. Knoten	bes. Kante	Typ
$\{d, e, f\}$	d	(d, e)	V
$\{d, e, f\}$	d	(d, f)	Q
$\{d, e, f\}$	d	-	-
$\{e, f\}$	e	(e, d)	R
$\{e, f\}$	e	-	-
$\{f\}$	f	(f, e)	Q
$\{f\}$	f	-	-
$\emptyset$			

Durchsuchen von Graphen

Satz

Falls der (un)gerichtete Graph G in Adjazenzlisten-Darstellung gegeben ist, durchläuft GraphSearch alle Knoten und Kanten von G in linearer Zeit $\mathcal{O}(n + m)$.

Beweis.

- Offensichtlich wird jeder Knoten u genau einmal zu B hinzugefügt.
- Dies geschieht zu dem Zeitpunkt, wenn u zum ersten Mal „besucht“ und das Feld `vis` für u auf `true` gesetzt wird.
- Außerdem werden in Zeile 6 von `Explore` alle von u ausgehenden Kanten durchlaufen, bevor u wieder aus B entfernt wird.
- Folglich werden tatsächlich alle Knoten und Kanten von G besucht.

Durchsuchen von Graphen

Beweis (Fortsetzung)

- Wir bestimmen nun die Laufzeit des Algorithmus GraphSearch.
- Innerhalb von Explore wird die while-Schleife für jeden Knoten v genau $(\deg(v) + 1)$ -mal bzw. $(\deg^+(v) + 1)$ -mal durchlaufen:
 - einmal für jeden Nachbarn w von v und
 - dann noch einmal, um v aus B zu entfernen.
- Insgesamt sind das $n + 2m$ Durchläufe im ungerichteten bzw. $n + m$ Durchläufe im gerichteten Fall.
- Bei Verwendung von Adjazenzlisten kann die nächste von einem Knoten v aus noch nicht besuchte Kante e in konstanter Zeit ermittelt werden, falls man für jeden Knoten v einen Zeiger auf e in der Adjazenzliste von v vorsieht.
- Die Gesamtlaufzeit des Algorithmus GraphSearch beträgt somit $\mathcal{O}(n + m)$.



Wege finden

Weitere Eigenschaften von Explore

- Falls beim Aufruf von Explore alle Knoten und Kanten als unbesucht markiert sind, liefert $\text{Explore}(w)$ zu allen von w aus erreichbaren Knoten v einen (gerichteten) w - v -Pfad.
- Dieser lässt sich mittels parent wie folgt zurückverfolgen.
- Sei

$$u_i = \begin{cases} v, & i = 0, \\ \text{parent}(u_{i-1}), & i > 0 \text{ und } u_{i-1} \neq \text{nil}. \end{cases}$$

und sei $\ell = \min\{i \geq 0 \mid u_{i+1} = \text{nil}\}$.

- Dann ist $u_\ell = w$ und $P = (u_\ell, \dots, u_0)$ ist ein (gerichteter) w - v -Pfad.
- Wir nennen P den **parent-Pfad** von v und bezeichnen ihn mit $P(v)$.

Wege finden

Satz

Falls beim Aufruf von `Explore` alle Knoten und Kanten als unbesucht markiert sind, berechnet `Explore(w)` zu allen erreichbaren Knoten v einen (gerichteten) w - v -Pfad $P(v)$.

Beweis.

Wir zeigen zuerst, dass `Explore(w)` alle von w aus erreichbaren Knoten besucht. Hierzu führen wir Induktion über die Länge ℓ eines kürzesten w - v -Weges.

$\ell = 0$: In diesem Fall ist $v = w$ und w wird in Zeile 1 besucht.

$\ell \rightsquigarrow \ell + 1$: Sei v ein Knoten mit Abstand $\ell + 1$ von w .

Dann hat ein Nachbarknoten $u \in N(v)$ den Abstand ℓ von w .
Folglich wird u nach IV besucht.

Da u erst dann aus B entfernt wird, wenn alle seine Nachbarn (bzw. Nachfolger) besucht wurden, wird auch v besucht.

Wege finden

Satz

Falls beim Aufruf von `Explore` alle Knoten und Kanten als unbesucht markiert sind, berechnet `Explore(w)` zu allen erreichbaren Knoten v einen (gerichteten) w - v -Pfad $P(v)$.

Beweis.

Es bleibt zu zeigen, dass `parent` einen Pfad $P(v)$ von w zu jedem besuchten Knoten v liefert. Hierzu führen wir Induktion über die Anzahl k der vor v besuchten Knoten.

$k = 0$: In diesem Fall ist $v = w$. Da `parent(w) = nil` ist, liefert `parent` einen w - v -Pfad (der Länge 0).

$k - 1 \rightsquigarrow k$: Sei $u = \text{parent}(v)$. Da u vor v besucht wird, liefert `parent` nach IV einen w - u -Pfad $P(u)$.

Wegen $u = \text{parent}(v)$ ist u der Entdecker von v und daher mit v durch eine Kante verbunden. Somit liefert `parent` auch für v einen w - v -Pfad $P(v)$. □

Zusammenhang in Graphen

Definition

- Ein Graph $G = (V, E)$ heißt **zusammenhängend**, falls es für alle Paare $\{u, v\} \in \binom{V}{2}$ einen u - v -Weg gibt.
- Sei $k \geq 1$. Ein Graph $G = (V, E)$ heißt **k -fach zusammenhängend**, falls G nach Entfernen von beliebigen $l \leq \min\{n - 1, k - 1\}$ Knoten immer noch zusammenhängend ist.

Bemerkung

- Es ist leicht zu sehen, dass die Relation

$$Z = \{(u, v) \in V \times V \mid \text{es gibt in } G \text{ einen } u\text{-}v\text{-Weg}\}$$

eine Äquivalenzrelation ist.

- Die durch die Äquivalenzklassen von Z induzierten Teilgraphen $G[U_1], \dots, G[U_k]$ heißen die **Zusammenhangskomponenten** (engl. **connected components**) von G .

Spannbäume und Spannwälder für ungerichtete Graphen

Definition

Sei $G = (V, E)$ ein Graph und $H = (U, F)$ ein Untergraph.

- H heißt **spannend**, falls $U = V$ ist.
- H ist ein **spannender Baum** (oder **Spannbaum**) von G , falls $U = V$ und H ein Baum ist.
- H ist ein **Spannwald** von G , falls $U = V$ und H ein Wald ist.

Bemerkung

- Es ist leicht zu sehen, dass für G genau dann ein Spannbaum existiert, wenn G zusammenhängend ist.
- Allgemeiner gilt, dass die Spannbäume für die Zusammenhangskomponenten von G einen Spannwald bilden.
- Dieser ist bzgl. der Subgraph-Relation maximal, da er in keinem größeren Spannwald enthalten ist.
- Ignorieren wir die Richtungen der Kanten im Suchwald W , so ist der resultierende Wald W' ein maximaler Spannwald für G .

Die Bäume des Suchwalds sind Spannbäume der Zusammenhangskomponenten

Da $\text{Explore}(w)$ alle von w aus erreichbaren Knoten findet, spannt jeder Baum des (ungerichteten) Suchwaldes $W' = (V, E'_{\text{parent}})$ mit

$$E_{\text{parent}} = \left\{ \{\text{parent}(v), v\} \mid v \in V \text{ und } \text{parent}(v) \neq \text{nil} \right\}$$

eine Zusammenhangskomponente von G .

Korollar

Sei G ein (ungerichteter) Graph.

- Der Algorithmus $\text{GraphSearch}(V, E)$ berechnet in Linearzeit einen Spannwald W' , dessen Bäume die Zusammenhangskomponenten von G spannen.
- Falls G zusammenhängend ist, ist W' ein Spannbaum für G .

Zusammenhangskomponenten in ungerichteten Graphen

Da $\text{Explore}(w)$ alle von w aus erreichbaren Knoten findet, lassen sich die Zusammenhangskomponenten eines ungerichteten Graphen durch folgende Variante von GraphSearch bestimmen.

Algorithmus $\text{CC}(V, E)$

```
1   $k := 0$ 
2  for all  $v \in V, e \in E$  do
3       $\text{cc}(v) := 0$ 
4       $\text{cc}(e) := 0$ 
5  for all  $w \in V$  do
6      if  $\text{cc}(w) = 0$  then
7           $k := k + 1$ 
8           $\text{ComputeCC}(k, w)$ 
```

Zusammenhangskomponenten in ungerichteten Graphen

Prozedur $\text{ComputeCC}(k, w)$

```
1   $\text{cc}(w) := k, \text{Init}(B), \text{Insert}(B, w)$ 
2  while  $\neg \text{Empty}(B)$  do
3       $u := \text{Element}(B)$ 
4      if  $\exists e = \{u, v\} \in E : \text{cc}(e) = 0$  then
5           $\text{cc}(e) := k$ 
6          if  $\text{cc}(v) = 0$  then
7               $\text{cc}(v) := k$ 
8               $\text{Insert}(B, v)$ 
9      else  $\text{Remove}(B)$ 
```

Korollar

Der Algorithmus $\text{CC}(V, E)$ bestimmt für einen ungerichteten Graphen $G = (V, E)$ in Linearzeit $\mathcal{O}(n + m)$ die Zusammenhangskomponenten $G_k = (V_k, E_k)$ von G , wobei $V_k = \{v \in V \mid \text{cc}(v) = k\}$ und $E_k = \{e \in E \mid \text{cc}(e) = k\}$ ist.

Breiten- und Tiefensuche

- Wir haben gesehen, dass $\text{Explore}(w)$ sowohl in Graphen als auch in Digraphen alle von w aus erreichbaren Knoten findet.
- Tatsächlich findet $\text{Explore}(w)$ zu allen von w aus erreichbaren Knoten sogar einen kürzesten Weg, falls wir die Datenstruktur B als Warteschlange Q implementieren.
- Die Benutzung einer Warteschlange bewirkt, dass zuerst alle Nachbarknoten $u_1, \dots, u_k$ des aktuellen Knotens u besucht werden, bevor ein anderer Knoten aktueller Knoten wird.
- Da die Suche also zuerst in die Breite geht, spricht man von einer **Breitensuche** (kurz **BFS**, engl. **breadth first search**).
- Den hierbei berechneten Suchwald bezeichnen wir als **Breitensuchwald**.
- Bei Benutzung eines Kellers wird dagegen u_1 aktueller Knoten, bevor die übrigen Nachbarknoten von u besucht werden.
- Daher führt die Benutzung eines Kellers zu einer **Tiefensuche** (kurz **DFS**, engl. **depth first search**).
- Der berechnete Suchwald heißt dann **Tiefensuchwald**.

Breiten- und Tiefensuche

Die Breitensuche eignet sich eher für Distanzprobleme wie z.B. das Finden

- kürzester Wege in Graphen und Digraphen,
- längster Wege in Bäumen (siehe Übungen) oder
- kürzester Wege in Distanzgraphen (Dijkstra-Algorithmus).

Dagegen liefert die Tiefensuche interessante Strukturinformationen wie z.B.

- die zweifachen Zusammenhangskomponenten in Graphen,
- die starken Zusammenhangskomponenten in Digraphen oder
- eine topologische Sortierung bei azyklischen Digraphen (s. Übungen).

Breitensuche

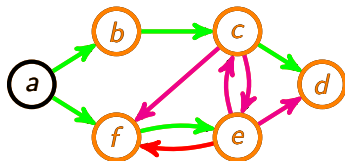
Algorithmus BFS(V, E)

```
1  for all  $v \in V, e \in E$  do
2     $\text{vis}(v) := \text{false}, \text{parent}(v) := \text{nil}, \text{vis}(e) := \text{false}$ 
3  for all  $w \in V$  do
4    if  $\text{vis}(w) = \text{false}$  then BFS-Explore( $w$ )
```

Prozedur BFS-Explore(w)

```
1   $\text{vis}(w) := \text{true}, \text{Init}(Q), \text{Enqueue}(Q, w)$ 
2  while  $\neg \text{Empty}(Q)$  do
3     $u := \text{Head}(Q)$ 
4    if  $\exists e = \{u, v\}$  bzw.  $e = (u, v) \in E : \text{vis}(e) = \text{false}$  then
5       $\text{vis}(e) := \text{true}$ 
6      if  $\text{vis}(v) = \text{false}$  then
7         $\text{vis}(v) := \text{true}, \text{parent}(v) := u, \text{Enqueue}(Q, v)$ 
8    else Dequeue( $Q$ )
```

Breitensuche in einem Digraphen



Schlange Q	bes. Knoten	bes. Kante	Typ
$\leftarrow a \leftarrow$	a	(a, b)	B
a, b	a	(a, f)	B
a, b, f	a	-	-
b, f	b	(b, c)	B
b, f, c	b	-	-
f, c	f	(f, e)	B
f, c, e	f	-	-
c, e	c	(c, d)	B

Q	bes. Knoten	bes. Kante	Typ
c, e, d	c	(c, e)	Q
c, e, d	c	(c, f)	Q
c, e, d	c	-	-
e, d	e	(e, c)	Q
e, d	e	(e, d)	Q
e, d	e	(e, f)	R
e, d	e	-	-
d	d	-	-

Kürzeste Wege

Satz

Sei G ein Graph oder Digraph und sei w Wurzel des von $\text{BFS-Explore}(w)$ berechneten Suchbaumes T . Dann liefert parent für jeden Knoten v in T einen kürzesten w - v -Weg $P(v)$.

Beweis.

Wir führen Induktion über die kürzeste Weglänge ℓ von w nach v in G .

$\ell = 0$: Dann ist $v = w$ und parent liefert einen Weg der Länge 0.

$\ell \rightsquigarrow \ell + 1$: Sei v ein Knoten, der den Abstand $\ell + 1$ von w in G hat.

Dann ex. ein Knoten $u \in N^-(v)$ mit Abstand ℓ von w in G .

Nach IV liefert also parent einen w - u -Weg $P(u)$ der Länge ℓ .

Da u erst aus Q entfernt wird, wenn alle Nachfolger von u entdeckt sind, wird v von u oder einem zuvor eingefügten Knoten z entdeckt.

Da Q als Schlange organisiert ist, ist $P(u)$ nicht kürzer als $P(z)$.

Daher folgt in beiden Fällen, dass $P(v)$ die Länge $\ell + 1$ hat. $\square$

Tiefensuche

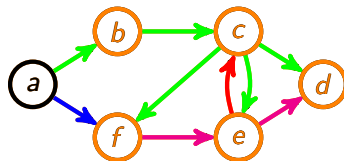
Algorithmus DFS(V, E)

```
1  for all  $v \in V, e \in E$  do
2     $\text{vis}(v) := \text{false}, \text{parent}(v) := \text{nil}, \text{vis}(e) := \text{false}$ 
3  for all  $w \in V$  do
4    if  $\text{vis}(w) = \text{false}$  then DFS-Explore( $w$ )
```

Prozedur DFS-Explore(w)

```
1   $\text{vis}(w) := \text{true}, \text{Init}(S), \text{Push}(S, w)$ 
2  while  $\neg \text{Empty}(S)$  do
3     $u := \text{Top}(S)$ 
4    if  $\exists e = \{u, v\}$  bzw.  $e = (u, v) \in E : \text{vis}(e) = \text{false}$  then
5       $\text{vis}(e) := \text{true}$ 
6      if  $\text{vis}(v) = \text{false}$  then
7         $\text{vis}(v) := \text{true}, \text{parent}(v) := u, \text{Push}(S, v)$ 
8    else
9       $\text{Pop}(S)$ 
```

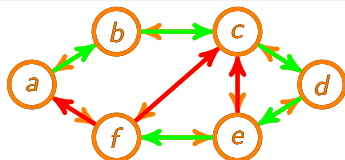
Tiefensuche in einem Digraphen



Keller S	bes. Knoten	bes. Kante	Typ
$a \leftrightarrow$	a	(a, b)	B
a, b	b	(b, c)	B
a, b, c	c	(c, d)	B
a, b, c, d	d	-	-
a, b, c	c	(c, e)	B
a, b, c, e	e	(e, c)	R
a, b, c, e	e	(e, d)	Q
a, b, c, e	e	-	-

S	bes. Knoten	bes. Kante	Typ
a, b, c	c	(c, f)	B
a, b, c, f	f	(f, e)	Q
a, b, c, f	f	-	-
a, b, c	c	-	-
a, b	b	-	-
a	a	(a, f)	V
a	a	-	-

Tiefensuche in einem Graphen



Keller S	Knoten	Kante	Typ	Keller S	Knoten	Kante	Typ
$a \leftrightarrow$	a	{a, b}	B	a, b, c, d, e, f	f	{f, c}	R
a, b	b	{b, a}	-	a, b, c, d, e, f	f	{f, e}	-
a, b	b	{b, c}	B	a, b, c, d, e, f	f	-	-
a, b, c	c	{c, b}	-	a, b, c, d, e	e	-	-
a, b, c	c	{c, d}	B	a, b, c, d	d	-	-
a, b, c, d	d	{d, c}	-	a, b, c	c	{c, e}	-
a, b, c, d	d	{d, e}	B	a, b, c	c	{c, f}	-
a, b, c, d, e	e	{e, c}	R	a, b, c	c	-	-
a, b, c, d, e	e	{e, d}	-	a, b	b	-	-
a, b, c, d, e	e	{e, f}	B	a	a	{a, f}	-
a, b, c, d, e, f	f	{f, a}	R	a	a	-	-

Tiefensuche (rekursiv)

- Die Tiefensuche lässt sich auch rekursiv implementieren.
- Dies hat den Vorteil, dass kein (expliziter) Keller benötigt wird.

Prozedur DFS-Explore-rec(w)

```
1 vis( $w$ ) := true
2 while  $\exists e = \{u, v\}$  bzw.  $e = (u, v) \in E : \text{vis}(e) = \text{false}$  do
3   vis( $e$ ) := true
4   if vis( $v$ ) = false then
5     parent( $v$ ) :=  $w$ 
6     DFS-Explore-rec( $v$ )
```

- Da DFS-Explore-rec(w) zu parent(w) zurückspringt, kann auch das Feld parent(w) als Keller fungieren.
- Daher lässt sich die Prozedur auch nicht-rekursiv ohne zusätzlichen Keller implementieren, indem die Rücksprünge explizit innerhalb einer Schleife ausgeführt werden (siehe Übungen).

Tiefensuche in Digraphen

- Bei der Tiefensuche lässt sich der Typ jeder Kante algorithmisch leicht bestimmen, wenn wir noch folgende Zusatzinformationen speichern.
- Ein neu entdeckter Knoten wird grau gefärbt. Wenn er abgearbeitet ist, wird er schwarz. Zu Beginn sind alle Knoten weiß.
- Zudem merken wir uns die Reihenfolge, in der die Knoten entdeckt werden, in einem Feld r .
- Dann lässt sich der Typ jeder Kante $e = (u, v)$ bei ihrem ersten Besuch wie folgt bestimmen:
 - Baumkante: $\text{farbe}(v) = \text{weiß}$,
 - Vorwärtskante: $\text{farbe}(v) \neq \text{weiß}$ und $r(v) \geq r(u)$,
 - Rückwärtskante: $\text{farbe}(v) = \text{grau}$ und $r(v) < r(u)$,
 - Querante: $\text{farbe}(v) = \text{schwarz}$ und $r(v) < r(u)$.
- Die folgende Variante von DFS berechnet diese Informationen.

Klassifikation der Kanten bei einer Tiefensuche

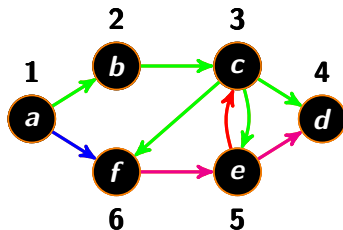
Algorithmus DFS(V, E)

```
1  r := 0
2  for all  $v \in V, e \in E$  do
3    farbe( $v$ ) := weiß
4    vis( $e$ ) := false
5  for all  $u \in V$  do
6    if farbe( $u$ ) = weiß then DFS-Explore( $u$ )
```

Prozedur DFS-Explore(u)

```
1  farbe( $u$ ) := grau
2  r := r + 1
3  r( $u$ ) := r
4  while  $\exists e = (u, v) \in E : \text{vis}(e) = \text{false}$  do
5    vis( $e$ ) := true
6    if farbe( $v$ ) = weiß then
7      DFS-Explore( $v$ )
8  farbe( $u$ ) := schwarz
```

Klassifikation der Kanten bei einer Tiefensuche



Keller	Farbe	Kante	Typ
a	$a \mapsto$ grau	(a, b)	B
a, b	$b \mapsto$ grau	(b, c)	B
a, b, c	$c \mapsto$ grau	(c, d)	B
a, b, c, d	$d \mapsto$ grau	-	-
	$d \mapsto$ schwarz		
a, b, c	-	(c, e)	B
a, b, c, e	$e \mapsto$ grau	(e, c)	R
a, b, c, e	-	(e, d)	Q

Keller	Farbe	Kante	Typ
a, b, c, e	$e \mapsto$ schwarz	-	-
a, b, c	-	(c, f)	B
a, b, c, f	$f \mapsto$ grau	(f, e)	Q
a, b, c, f	$f \mapsto$ schwarz	-	-
a, b, c	$c \mapsto$ schwarz	-	-
a, b	$b \mapsto$ schwarz	-	-
a	-	(a, f)	V
a	$a \mapsto$ schwarz	-	-

Tiefensuche in ungerichteten Graphen

- Bei der Tiefensuche in ungerichteten Graphen können weder Quer- noch Vorwärtskanten auftreten.
- Da v beim ersten Besuch einer solchen Kante (u, v) nicht weiß ist und alle grauen Knoten auf dem parent-Pfad $P(u)$ liegen, müsste v nämlich bereits schwarz sein.
- Dies ist aber nicht möglich, da die Kante $\{u, v\}$ in v - u -Richtung noch gar nicht durchlaufen wurde.
- Folglich sind alle Kanten, die nicht zu einem neuen Knoten führen, Rückwärtskanten.
- Das Fehlen von Quer- und Vorwärtskanten spielt bei manchen Anwendungen eine wichtige Rolle, etwa bei der Zerlegung eines Graphen G in seine **zweifachen Zusammenhangskomponenten**.

Zusammenhang in Digraphen

Definition

- Ein Digraph $G = (V, E)$ heißt **schwach zusammenhängend**, wenn es für jedes Paar $\{u, v\} \in \binom{V}{2}$ einen gerichteten u - v -Pfad oder einen gerichteten v - u -Pfad gibt.
- $G = (V, E)$ heißt **stark zusammenhängend**, wenn es für jedes Paar $\{u, v\} \in \binom{V}{2}$ sowohl einen gerichteten u - v -Pfad als auch einen gerichteten v - u -Pfad gibt.

Starke Zusammenhangskomponenten in Digraphen

Bemerkung

Sei $G = (V, E)$ ein Digraph. Dann ist leicht zu sehen, dass die Relation

$$S = \{(u, v) \in V \times V \mid \text{in } G \text{ ex. ein } u\text{-}v\text{-Weg und ein } v\text{-}u\text{-Weg}\}$$

eine Äquivalenzrelation ist. Für $(u, v) \in S$ schreiben wir auch kurz $u \sim v$.

Definition

Die durch die Äquivalenzklassen $U_1, \dots, U_k$ von S induzierten Teilgraphen $G[U_1], \dots, G[U_k]$ heißen die **starken Zusammenhangskomponenten** (engl. **strongly connected components**) von G .

Satz

Sei $G = (V, E)$ ein Digraph mit den starken Zusammenhangskomponenten $G[U_1], \dots, G[U_k]$. Dann ist der Digraph (C, D) mit $C = \{1, \dots, k\}$ und

$$D = \{(i, j) \mid 1 \leq i \neq j \leq k \wedge \exists u \in U_i, v \in U_j : (u, v) \in E\}$$

azyklisch.

Starke Zusammenhangskomponenten in Digraphen

Satz

Sei $G = (V, E)$ ein Digraph mit den starken Zusammenhangskomponenten $G[U_1], \dots, G[U_k]$. Dann ist der Digraph (C, D) mit $C = \{1, \dots, k\}$ und

$$D = \{(i, j) \mid 1 \leq i \neq j \leq k \wedge \exists u \in U_i, v \in U_j : (u, v) \in E\}$$

azyklisch.

Beweis.

- Da der Digraph (C, D) schlingenfrei ist, müsste ein Zyklus mindestens zwei verschiedene Knoten $i \neq j$ enthalten.
- Dann würden aber die beiden starken Zusammenhangskomponenten $G[U_i]$ und $G[U_j]$ zu einer einzigen Komponente verschmelzen. □

Starke Zusammenhangskomponenten in Digraphen

- Sei $G = (V, E)$ ein Digraph mit zugehörigem Tiefensuchwald W und starken Zusammenhangskomponenten $G[U_1], \dots, G[U_k]$.
- Den im Suchwald W durch alle Nachfahren von v induzierten Baum mit der Wurzel v bezeichnen wir mit $T(v)$.
- Für $i = 1, \dots, k$ sei s_i der erste bei der Tiefensuche innerhalb von U_i besuchte Knoten. Wir bezeichnen s_i als den **Startknoten** von U_i .
- Dann bilden die Knoten $s_1, \dots, s_k$ ein Repräsentantensystem für die Äquivalenzklassen $U_1, \dots, U_k$.
- Sei V_i die Menge aller Knoten in $T(s_i)$, die für keinen echten Nachfahren s_j von s_i in $T(s_j)$ enthalten sind.
- Es ist klar, dass die Mengen $V_1, \dots, V_k$ eine Partition von V bilden und s_i in $V_i \cap U_i$ enthalten ist.
- Wir zeigen, dass $V_i \cap U_j$ nur im Fall $i = j$ nicht leer ist (was $V_i = U_i$ für alle i impliziert).

Starke Zusammenhangskomponenten in Digraphen

- Wir zeigen, dass $V_i \cap U_j$ nur im Fall $i = j$ nicht leer ist (was $V_i = U_i$ für alle i impliziert). Sei also $u \in V_i \cap U_j$.
- Da s_j erst schwarz wird, nachdem alle von s_j aus erreichbaren Knoten besucht wurden, enthält $T(s_j)$ alle Knoten in U_j , insbesondere also u .
- Wegen $u \in V_i$ ist u für keinen echten Nachfahren s_k von s_i in $T(s_k)$ enthalten, d.h. s_j kann wegen $u \in T(s_j)$ kein echter Nachfahre von s_i sein.
- Da u wegen $u \in V_i$ in $T(s_i)$ und zudem in $T(s_j)$ liegt, muss s_j also ein Vorfahre von s_i sein.
- Wegen $u \sim s_j$ impliziert dies jedoch $u \sim s_i$ und somit $i = j$.
- Die Mengen U_i lassen sich also leicht bestimmen, falls wir die Startknoten s_i während der Tiefensuche identifizieren können.

Starke Zusammenhangskomponenten in Digraphen

- Die Mengen U_i lassen sich also leicht bestimmen, falls wir die Startknoten s_i bei ihrem letzten Besuch identifizieren können.
- In diesem Fall gehören alle seit dem ersten Besuch von s_i besuchten Knoten zu U_i , die bis zum letzten Besuch von s_i nicht schon einer anderen Komponente zugeordnet wurden.
- Um die Startknoten erkennen zu können, betrachten wir die Funktion

$$\text{low}(u) = \min\{r(v) \mid v \in P(u) \text{ und es gibt einen } u\text{-}v\text{-Weg in } G\}.$$

- Da der zu u gehörige Startknoten $s_i \sim u$ von u aus erreichbar ist und auf dem parent-Pfad $P(u)$ von u liegt, folgt $\text{low}(u) \leq r(s_i) \leq r(u)$.
- Da es keinen Weg von u zu einem Knoten $v \in P(u)$ mit $r(v) < r(s_i)$ geben kann (sonst wäre v Startknoten von U_i), ist $\text{low}(u) = r(s_i)$.
- Folglich ist

$$\text{low}(u) = \min\{r(s_i) \mid s_i \in P(u) \text{ und es gibt einen } u\text{-}s_i\text{-Weg in } G\}$$

und nur für die Startknoten nehmen low und r den gleichen Wert an.

Starke Zusammenhangskomponenten in Digraphen

- Folglich ist

$$\text{low}(u) = \min\{r(s_i) \mid s_i \in P(u) \text{ und es gibt einen } u\text{-}s_i\text{-Weg in } G\}$$

und nur für die Startknoten nehmen low und r den gleichen Wert an.

- Einfacher als low lässt sich die Funktion

$$l(u) = \min\{r(v) \mid v = u \vee \exists u' \in T(u), v \in T(s_i) : (u', v) \in E\},$$

berechnen, wobei s_i der eindeutig bestimmte Startknoten mit $u \sim s_i$ ist.

- Dann gilt $\text{low}(u) \leq l(u) \leq r(u)$, wobei $l(u) = r(u)$ mit $\text{low}(u) = r(u)$ (also mit $u = s_i$) gleichbedeutend ist.
- Ist nämlich $u \neq s_i$, so liegt s_i nicht in $T(u)$ und daher muss jeder $u\text{-}s_i$ -Weg eine (Rückwärts- oder Quer-) Kante (u', v) mit $u' \in T(u)$ und $v \in T(s_i) - T(u)$ enthalten.
- Da jedoch alle von u aus erreichbaren Knoten, deren r -Wert größer als der von u ist, in $T(u)$ enthalten sind, muss $r(v) < r(u)$ und somit $l(u) < r(u)$ sein.

Starke Zusammenhangskomponenten in Digraphen

- Der folgende Algorithmus SCC berechnet für jeden Knoten u den Wert $l(u)$ und gibt der Reihe nach die Mengen U_i aus.
- Dass SCC bis zum letzten Besuch eines Knotens u tatsächlich den Wert $l(u)$ korrekt berechnet, lässt sich induktiv über die Anzahl der zuvor abgearbeiteten Knoten zeigen.
- SCC benutzt einen Keller S , um darin alle entdeckten Knoten, die noch keiner Menge U_i zugeordnet werden konnten, zu speichern.
- Das Feld `onstack` speichert die Information, welche Knoten aktuell im Keller S enthalten sind.
- Besitzt ein Knoten u bei seinem letzten Besuch den Wert $l(u) = r(u)$, so wird die Prozedur **Output-SCC(u)** aufgerufen.
- **Output-SCC(u)** leert den Keller S bis einschließlich u und gibt diese Knoten als neu entdeckte Menge U_i aus.

Starke Zusammenhangskomponenten in Digraphen

Algorithmus $\text{SCC}(V, E)$

```
1   $r := 0$ ,  $\text{Init}(S)$ 
2  for all  $v \in V, e \in E$  do  $r(v) := 0$ ,  $\text{onstack}(v) := \text{vis}(e) := \text{false}$ 
3  for all  $u \in V$  do if  $r(u) = 0$  then  $\text{Compute-SCC}(u)$ 
```

Prozedur $\text{Compute-SCC}(u)$

```
1   $r := r + 1$ ,  $r(u) := \textcolor{brown}{l(u)} := \textcolor{brown}{r}$ ,  $\text{Push}(S, u)$ ,  $\text{onstack}(u) := \text{true}$ 
2  while  $\exists e = (u, v) \in E : \text{vis}(e) = \text{false}$  do
3     $\text{vis}(e) := \text{true}$ 
4    if  $r(v) = 0$  then  $\text{Compute-SCC}(v)$ ,  $\textcolor{brown}{l(u)} := \min\{\textcolor{brown}{l(u)}, \textcolor{brown}{l(v)}\}$ 
5    else if  $\text{onstack}(v) = \text{true}$  then  $\textcolor{brown}{l(u)} := \min\{\textcolor{brown}{l(u)}, r(v)\}$ 
6  if  $\textcolor{brown}{l(u)} = r(u)$  then  $\text{Output-SCC}(u)$ 
```

Prozedur $\text{Output-SCC}(u)$

```
1  write(Neue Komponente: )
2  repeat  $v := \text{Pop}(S)$ ,  $\text{onstack}(v) := \text{false}$ , write( $v$ )
3  until( $v = u$ )
```

Schnittpunkte in Graphen

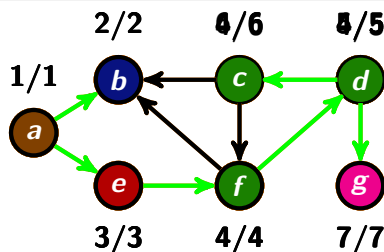
Algorithmus $\text{Cutpoints}(V, E)$

```
1  for all  $v \in V, e \in E$  do  $r(v) := 0, \text{vis}(e) := \text{false}$ 
2  for all  $u \in V$  do if  $r(u) = 0$  then
     $r := c := 0, \text{Compute-Cutpoints}(u)$ 
```

Prozedur $\text{Compute-Cutpoints}(u)$

```
1   $r := r + 1, r(u) := l(u) := r$ 
2  while  $\exists e = (u, v) \in E : \text{vis}(e) = \text{false}$  do
3       $\text{vis}(e) := \text{true}$ 
4      if  $r(u) = 1$  then  $c := c + 1$ 
5      if  $r(v) = 0$  then
6           $\text{Compute-Cutpoints}(v),$ 
7           $l(u) := \min\{l(u), l(v)\}$ 
8          if  $l(v) \geq r(u)$  then  $\text{Output}(u)$ 
9          else  $l(u) := \min\{l(u), r(v)\}$ 
10 if  $r(u) = 1 \wedge c \geq 2$  then  $\text{Output}(u)$ 
```

Starke Zusammenhangskomponenten in Digraphen



S	u	$\ell(u)$	$r(u)$	(u, v)	out
a	a	1	1	(a, b)	-
a, b	b	2	2	-	b
a	a	1	1	(a, e)	-
a, e	e	3	3	(e, f)	-
a, e, f	f	4	4	(f, b)	-
a, e, f	f	4	4	(f, d)	-
a, e, f, d	d	5	5	(d, c)	-
a, e, f, d, c	c	6	6	(c, b)	-

S	u	$\ell(u)$	$r(u)$	(u, v)	out
a, e, f, d, c	c	6	6	(c, f)	-
a, e, f, d, c	c	4	6	-	-
a, e, f, d, c	d	4	5	(d, g)	-
a, e, f, d, c, g	g	7	7	-	g
a, e, f, d, c	d	4	5	-	-
a, e, f, d, c	f	4	4	-	f, d, c
a, e	e	3	3	-	e
a	a	1	1	-	a

Kürzeste Wege in Distanzgraphen

- In vielen Anwendungen tritt das Problem auf, einen kürzesten Weg von einem Startknoten s zu einem Zielknoten t in einem Digraphen zu finden, dessen Kanten (u, v) vorgegebene **Längen** $\ell(u, v) \geq 0$ haben.
- Die Länge eines Weges $W = (v_0, \dots, v_j)$ ist $\ell(W) = \sum_{i=0}^{j-1} \ell(v_i, v_{i+1})$.
- Die kürzeste Weglänge von s nach t wird als **Distanz** $\text{dist}(s, t)$ zwischen s und t bezeichnet,

$$\text{dist}(s, t) = \min\{\ell(W) \mid W \text{ ist ein } s\text{-}t\text{-Weg}\}.$$

Falls kein s - t -Weg existiert, setzen wir $\text{dist}(s, t) = \infty$.

Der Dijkstra-Algorithmus

- Der Dijkstra-Algorithmus findet einen kürzesten Weg $P(u)$ von s zu allen erreichbaren Knoten u (single-source shortest-path problem).
- Während der Suche werden alle Knoten u , zu denen bereits ein s - u -Weg $P(u)$ bekannt ist, zusammen mit der Weglänge g in einer Menge U gespeichert und erst dann wieder entfernt, wenn $P(u)$ optimal ist.
- Damit die Suche effizient ist, sollten folgende Operationen auf der Datenstruktur U effizient ausführbar sein.

Benötigte Operationen

Init(U): Initialisiert U als leere Menge.

Update(U, u, g): Erniedrigt den Wert von u auf g (nur wenn der aktuelle Wert größer als g ist). Ist u noch nicht in U enthalten, wird u mit dem Wert g zu U hinzugefügt.

RemoveMin(U): Gibt ein Element aus U mit dem kleinsten Wert zurück und entfernt es aus U (ist U leer, wird der Wert `nil` zurückgegeben).

Der Dijkstra-Algorithmus

- Voraussetzung für die Korrektheit des Algorithmus' ist, dass alle Kanten eine nichtnegative Länge $\ell(u, v) \geq 0$ haben.
- In diesem Fall wird $D = (V, E, \ell)$ auch **Distanzgraph** genannt.
- Während der Suche werden bestimmte Kanten $e = (u, v)$ daraufhin getestet, ob $g(u) + \ell(u, v) < g(v)$ ist.
- Da in diesem Fall die Kante e auf eine Herabsetzung von $g(v)$ auf den Wert $g(u) + \ell(u, v)$ „drängt“, wird diese Wertzuweisung als **Relaxation** von e bezeichnet.
- Welche Kanten (u, v) auf Relaxation getestet werden, wird beim Dijkstra- Algorithmus durch eine einfache **Greedystrategie** bestimmt:
Wähle u unter allen noch nicht fertigen Knoten mit minimalem g -Wert und teste alle Kanten (u, v) , für die v nicht schon fertig ist.
- Der Algorithmus führt also eine modifizierte **Breitensuche** aus, bei der die in Bearbeitung befindlichen Knoten in einer **Prioritätswarteschlange** verwaltet werden.

Der Dijkstra-Algorithmus

Algorithmus Dijkstra(V, E, ℓ, s)

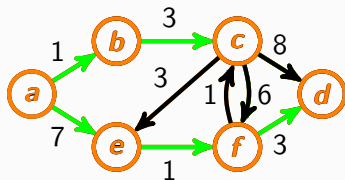
```
1  for all  $v \in V$  do
2       $g(v) := \infty$ ,  $\text{parent}(v) := \text{nil}$ ,  $\text{done}(v) := \text{false}$ 
3  Init( $U$ ), Update( $U, s, 0$ ),  $g(s) := 0$ 
4  while  $u := \text{RemoveMin}(U) \neq \text{nil}$  do
5       $\text{done}(u) := \text{true}$ 
6      for all  $v \in N^+(u)$  do
7          if  $\text{done}(v) = \text{false} \wedge g(u) + \ell(u, v) < g(v)$  then
8               $g(v) := g(u) + \ell(u, v)$ , Update( $U, v, g(v)$ ),  $\text{parent}(v) := u$ 
```

- $g(u)$ speichert die aktuelle Länge des Pfades $P(u)$. Knoten außerhalb des aktuellen Breitensuchbaums T haben den Wert ∞ .
- In jedem Schleifendurchlauf wird in Zeile 4 ein Knoten u mit minimalem g -Wert aus U entfernt und als fertig markiert.
- Anschließend wird für alle von u wegführenden Kanten $e = (u, v)$ geprüft, ob eine Relaxation ansteht.
- Wenn ja, wird U aktualisiert und e wird neue Baumkante in T .

Der Dijkstra-Algorithmus

Beispiel

Betrachte folgenden Distanzgraphen mit dem Startknoten a :



Inhalt von U	entfernt	besuchte Kanten	Update-Operationen
$(a, 0)$	$(a, 0)$	$(a, b), (a, e)$	$(b, 1), (e, 7)$
$(b, 1), (e, 7)$	$(b, 1)$	(b, c)	$(c, 4)$
$(c, 4), (e, 7)$	$(c, 4)$	$(c, d), (c, e), (c, f)$	$(d, 12), (f, 10)$
$(e, 7), (f, 10), (d, 12)$	$(e, 7)$	(e, f)	$(f, 8)$
$(f, 8), (d, 12)$	$(f, 8)$	$(f, c), (f, d)$	$(d, 11)$
$(d, 11)$	$(d, 11)$	—	—

Korrektheit des Dijkstra-Algorithmus'

Algorithmus $\text{Dijkstra}(V, E, \ell, s)$

```
1  for all  $v \in V$  do
2     $g(v) := \infty$ ,  $\text{parent}(v) := \text{nil}$ ,  $\text{done}(v) := \text{false}$ 
3   $g(s) := 0$ ,  $\text{Init}(U)$ ,  $\text{Update}(U, s, 0)$ 
4  while  $u := \text{RemoveMin}(U) \neq \text{nil}$  do
5     $\text{done}(u) := \text{true}$ 
6    for all  $v \in N^+(u)$  do
7      if  $\text{done}(v) = \text{false} \wedge g(u) + \ell(u, v) < g(v)$  then
8         $g(v) := g(u) + \ell(u, v)$ ,  $\text{Update}(U, v, g(v))$ ,  $\text{parent}(v) := u$ 
```

Satz

Sei $D = (V, E, \ell)$ ein Distanzgraph und sei $s \in V$. Dann berechnet $\text{Dijkstra}(V, E, \ell, s)$ für alle von s aus erreichbaren Knoten $t \in V$ einen kürzesten s - t -Weg $P(t)$.

Korrektheit des Dijkstra-Algorithmus'

Satz

Sei $D = (V, E, \ell)$ ein Distanzgraph und sei $s \in V$. Dann berechnet $\text{Dijkstra}(V, E, \ell, s)$ für alle von s aus erreichbaren Knoten $t \in V$ einen kürzesten s - t -Weg $P(t)$.

Beweis.

- Wir zeigen zuerst, dass alle von s aus erreichbaren Knoten $t \in V$ zu U hinzugefügt werden.
- Dies folgt aus der Tatsache, dass s zu U hinzugefügt wird, und spätestens dann, wenn ein Knoten u in Zeile 4 aus U entfernt wird, sämtliche Nachfolger von u zu U hinzugefügt werden.
- Zudem ist klar, dass $g(t) \geq \text{dist}(s, t)$ ist, da $P(t)$ im Fall $g(t) < \infty$ ein s - t -Weg der Länge $g(t)$ ist.
- Als nächstes zeigen wir, dass $P(t)$ für jeden aus U entfernten Knoten t ein kürzester s - t -Weg ist, d.h. es gilt $g(t) \leq \text{dist}(s, t)$.

Korrektheit des Dijkstra-Algorithmus'

- Wir zeigen induktiv über die Anzahl k der vor u aus U entfernten Knoten, dass $g(u) \leq \text{dist}(s, u)$ ist.

$k = 0$: In diesem Fall ist $u = s$ und $P(u)$ hat die Länge $g(u) = 0$.

$k - 1 \rightsquigarrow k$: Sei $W = v_0, \dots, v_j = u$ ein kürzester s - u -Weg in G und sei v_i der Knoten mit maximalem Index i , der vor u aus U entfernt wird. Nach IV gilt dann

$$g(v_i) = \text{dist}(s, v_i). \quad (1)$$

Zudem ist

$$g(v_{i+1}) \leq g(v_i) + \ell(v_i, v_{i+1}). \quad (2)$$

Da u im Fall $u \neq v_{i+1}$ vor v_{i+1} aus U entfernt wird, ist

$$g(u) \leq g(v_{i+1}). \quad (3)$$

Daher folgt

$$\begin{aligned} g(u) &\stackrel{(3)}{\leq} g(v_{i+1}) \stackrel{(2)}{\leq} g(v_i) + \ell(v_i, v_{i+1}) \\ &\stackrel{(1)}{=} \text{dist}(s, v_i) + \ell(v_i, v_{i+1}) \\ &= \text{dist}(s, v_{i+1}) \leq \text{dist}(s, u). \end{aligned}$$

Laufzeit des Dijkstra-Algorithmus'

Algorithmus Dijkstra(V, E, ℓ, s)

```
1  for all  $v \in V$  do
2       $g(v) := \infty$ ,  $\text{parent}(v) := \text{nil}$ ,  $\text{done}(v) := \text{false}$ 
3   $g(s) := 0$ ,  $\text{Init}(U)$ ,  $\text{Update}(U, s, 0)$ 
4  while  $u := \text{RemoveMin}(U) \neq \text{nil}$  do
5       $\text{done}(u) := \text{true}$ 
6      for all  $v \in N^+(u)$  do
7          if  $\text{done}(v) = \text{false} \wedge g(u) + \ell(u, v) < g(v)$  then
8               $g(v) := g(u) + \ell(u, v)$ ,  $\text{Update}(U, v, g(v))$ ,  $\text{parent}(v) := u$ 
```

- Wir überlegen uns zuerst, wie oft die einzelnen Operationen Init , RemoveMin und Update auf der Datenstruktur U ausgeführt werden.
- Sei $n = \|V\|$ die Anzahl der Knoten und $m = \|E\|$ die Anzahl der Kanten des Eingabegraphen.

Laufzeit des Dijkstra-Algorithmus'

Algorithmus Dijkstra(V, E, ℓ, s)

```
1  for all  $v \in V$  do
2     $g(v) := \infty$ ,  $\text{parent}(v) := \text{nil}$ ,  $\text{done}(v) := \text{false}$ 
3   $g(s) := 0$ ,  $\text{Init}(U)$ ,  $\text{Update}(U, s, 0)$ 
4  while  $u := \text{RemoveMin}(U) \neq \text{nil}$  do
5     $\text{done}(u) := \text{true}$ 
6    for all  $v \in N^+(u)$  do
7      if  $\text{done}(v) = \text{false} \wedge g(u) + \ell(u, v) < g(v)$  then
8         $g(v) := g(u) + \ell(u, v)$ ,  $\text{Update}(U, v, g(v))$ ,  $\text{parent}(v) := u$ 
```

- Die Init-Operation wird nur einmal ausgeführt.
- Da die while-Schleife für jeden von s aus erreichbaren Knoten genau einmal durchlaufen wird, wird die RemoveMin-Operation höchstens $\min\{n, m\}$ -mal ausgeführt.
- Da der Dijkstra-Algorithmus jede Kante höchstens einmal besucht, wird die Update-Operation höchstens m -mal ausgeführt.

Laufzeit des Dijkstra-Algorithmus'

- Die *Init*-Operation wird nur einmal ausgeführt.
- Da die *while*-Schleife für jeden von *s* aus erreichbaren Knoten genau einmal durchlaufen wird, wird die *RemoveMin*-Operation höchstens $\min\{n, m\}$ - mal ausgeführt.
- Da der Dijkstra-Algorithmus jede Kante höchstens einmal besucht, wird die *Update*-Operation höchstens m -mal ausgeführt.
- Bezeichne *Init*(n), *RemoveMin*(n) und *Update*(n) den Aufwand zum Ausführen der Operationen *Init*, *RemoveMin* und *Update* für den Fall, dass U nicht mehr als n Elemente aufzunehmen hat.
- Dann ist die Laufzeit des Dijkstra-Algorithmus' also durch

$$\mathcal{O}(n + m + \textit{Init}(n) + \min\{n, m\} \cdot \textit{RemoveMin}(n) + m \cdot \textit{Update}(n))$$

beschränkt.

Laufzeit des Dijkstra-Algorithmus'

- Dann ist die Laufzeit des Dijkstra-Algorithmus' also durch

$$\mathcal{O}(n + m + \text{Init}(n) + \min\{n, m\} \cdot \text{RemoveMin}(n) + m \cdot \text{Update}(n))$$

beschränkt.

- Die Laufzeit des Dijkstra-Algorithmus' hängt also wesentlich davon ab, wie wir die Datenstruktur U implementieren.
- Falls alle Kanten die gleiche Länge haben, wachsen die Distanzwerte der Knoten monoton in der Reihenfolge ihres Besuchs.
- D.h. wir können U in diesem Fall als Warteschlange implementieren.
- Dies führt wie bei der Prozedur BFS-Explore auf eine Laufzeit von $\mathcal{O}(n + m)$.
- Für den allgemeinen Fall, dass die Kanten unterschiedliche Längen haben, betrachten wir folgende drei Möglichkeiten.

Laufzeit des Dijkstra-Algorithmus'

- Dann ist die Laufzeit des Dijkstra-Algorithmus' also durch

$$\mathcal{O}(n + m + \text{Init}(n) + \min\{n, m\} \cdot \text{RemoveMin}(n) + m \cdot \text{Update}(n))$$

beschränkt.

Für den allgemeinen Fall, dass die Kanten unterschiedliche Längen haben, betrachten wir folgende drei Möglichkeiten.

- 1 Da die Felder g und $done$ bereits alle nötigen Informationen enthalten, kann man auf die (explizite) Implementierung von U auch verzichten.

In diesem Fall kostet die RemoveMin -Operation allerdings Zeit $\mathcal{O}(n)$, was auf eine Gesamtlaufzeit von $\mathcal{O}(n^2)$ führt.

Dies ist asymptotisch optimal, wenn G relativ dicht ist, also $m = \Omega(n^2)$ Kanten enthält.

Ist G dagegen relativ dünn, d.h. $m = o(n^2)$, so empfiehlt es sich, U als Prioritätswarteschlange zu implementieren.

Laufzeit des Dijkstra-Algorithmus'

- ② Es ist naheliegend, U in Form eines Heaps H zu implementieren. In diesem Fall lässt sich die Operation `RemoveMin` in Zeit $\mathcal{O}(\log n)$ implementieren.

Da die Prozedur `Update` einen linearen Zeitaufwand erfordert, ist es effizienter, sie durch eine `Insert`-Operation zu simulieren.

Dies führt zwar dazu, dass derselbe Knoten evtl. mehrmals mit unterschiedlichen Werten in H gespeichert wird.

Die Korrektheit bleibt aber dennoch erhalten, wenn wir nur die erste Entnahme eines Knotens aus H beachten und die übrigen ignorieren.

Da für jede Kante höchstens ein Knoten in H eingefügt wird, erreicht H maximal die Größe n^2 und daher sind die Heap-Operationen `Insert` und `RemoveMin` immer noch in Zeit $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$ ausführbar.

Insgesamt erhalten wir somit eine Laufzeit von $\mathcal{O}(n + m \log n)$.

Dies ist zwar für dünne Graphen sehr gut, aber für dichte Graphen schlechter als die implizite Implementierung von U mittels `g` und `done`.

Laufzeit des Dijkstra-Algorithmus'

- ③ Als weitere Möglichkeit kann U auch in Form eines so genannten **Fibonacci-Heaps** F implementiert werden.

Dieser benötigt nur eine konstante amortisierte Laufzeit $\mathcal{O}(1)$ für die Update-Operation und $\mathcal{O}(\log n)$ für die RemoveMin-Operation.

Insgesamt führt dies auf eine Laufzeit von $\mathcal{O}(m + n \log n)$.

Allerdings sind Fibonacci-Heaps erst bei sehr großen Graphen mit mittlerer Dichte schneller.

Wir fassen zusammen:

	implizit	Heap	Fibonacci-Heap
Init	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Update	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
RemoveMin	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Gesamtlaufzeit	$\mathcal{O}(n^2)$	$\mathcal{O}(n + m \log n)$	$\mathcal{O}(m + n \log n)$

Offen ist, ob eine Verbesserung auf $\mathcal{O}(n + m)$ möglich ist.

Kürzeste Pfade in Graphen mit negativen Kantengewichten

- In manchen Anwendungen treten negative Kantengewichte auf.
- Geben die Kantengewichte beispielsweise die mit einer Kante verbundenen Kosten wider, so kann ein Gewinn durch negative Kosten modelliert werden.
- Gesucht ist dann meist ein kürzester Pfad (also ein kürzester Weg, der jeden Knoten höchstens einmal besucht).
- Auf diese Weise lassen sich auch längste Pfade in Distanzgraphen berechnen, indem man alle Kantenlängen $\ell(u, v)$ mit -1 multipliziert und in dem resultierenden Graphen einen kürzesten Pfad bestimmt.
- Die Komplexität des Problems hängt wesentlich davon ab, ob man gerichtete Kreise mit negativer Länge (bzw. Kosten) zulässt oder nicht.
- Falls negative Kreise zugelassen werden, ist das Problem NP-hart.
- Andernfalls existieren effiziente Algorithmen wie z.B.
 - der Bellman-Ford-Algorithmus (BF-Algorithmus) oder
 - der Bellman-Ford-Moore-Algorithmus (BFM-Algorithmus).

Der Ford-Algorithmus

- Der Ford-Algorithmus arbeitet ganz ähnlich wie der Dijkstra-Algorithmus, betrachtet aber jede Kante nicht wie dieser nur einmal, sondern eventuell mehrmals.
- In seiner einfachsten Form sucht der Algorithmus wiederholt eine Kante $e = (u, v)$ mit

$$g(u) + \ell(u, v) < g(v)$$

und aktualisiert den Wert von $g(v)$ auf $g(u) + \ell(u, v)$ (Relaxation).

- Die Laufzeit hängt dann wesentlich davon ab, in welcher Reihenfolge die Kanten auf Relaxation getestet werden.
- Im besten Fall lässt sich eine lineare Laufzeit erreichen (z.B. wenn der zugrunde liegende Digraph azyklisch ist).
- Bei der Bellman-Ford-Variante wird in $\mathcal{O}(nm)$ Schritten ein kürzester Weg von s zu allen erreichbaren Knoten gefunden (sofern keine negativen Kreise existieren).

Der Ford-Algorithmus

- Wir zeigen induktiv über die Anzahl k der Kanten eines kürzesten s - u -Weges, dass $g(u) = \text{dist}(s, u)$ gilt, falls g für alle Kanten (u, v) die Dreiecksungleichung $g(v) \leq g(u) + \ell(u, v)$ erfüllt (also keine Relaxationen mehr möglich sind).
- Im Fall $k = 0$ ist nämlich $u = s$ und somit $g(s) = 0 = \text{dist}(s, s)$.
- Im Fall $k > 0$ sei v ein Knoten, dessen kürzester s - v -Weg W aus k Kanten besteht.
- Dann gilt nach IV für den Vorgänger u von v auf W $g(u) = \text{dist}(s, u)$.
- Aufgrund der Dreiecksungleichung folgt dann

$$g(v) \leq g(u) + \ell(u, v) = \text{dist}(s, u) + \ell(u, v) = \text{dist}(s, v).$$

- Aus dem Beweis folgt zudem, dass $g(v)$ nach Relaxation aller Kanten eines kürzesten s - v -Weges W (in der Reihenfolge, in der die Kanten in W durchlaufen werden) den Wert $\text{dist}(s, v)$ hat.
- Dies gilt auch für den Fall, dass zwischendurch noch weitere Kantenrelaxationen stattfinden.

Der Bellman-Ford-Algorithmus

- Diese Variante prüft in $n - 1$ Iterationen jeweils alle Kanten auf Relaxation.
- Sind in der n -ten Runde noch weitere Relaxationen möglich, muss ein negativer Kreis existieren.
- Die Laufzeit ist offensichtlich $\mathcal{O}(nm)$ und die Korrektheit folgt leicht durch Induktion über die minimale Anzahl von Kanten eines kürzesten s - t -Weges.
- Zudem wird bei jeder Relaxation einer Kante (u, v) der Vorgänger u im Feld $\text{parent}(v)$ vermerkt, so dass sich ein kürzester Pfad von s zu allen erreichbaren Knoten (bzw. ein negativer Kreis) rekonstruieren lässt.

Der Bellman-Ford-Algorithmus

Algorithmus $\text{BF}(V, E, \ell, s)$

```
1  for all  $v \in V$  do
2       $g(v) := \infty$ 
3       $\text{parent}(v) := \text{nil}$ 
4   $g(s) := 0$ 
5  for  $i := 1$  to  $n - 1$  do
6      for all  $(u, v) \in E$  do
7          if  $g(u) + \ell(u, v) < g(v)$  then
8               $g(v) := g(u) + \ell(u, v)$ 
9               $\text{parent}(v) := u$ 
10 for all  $(u, v) \in E$  do
11     if  $g(u) + \ell(u, v) < g(v)$  then
12         error(es gibt einen negativen Kreis)
```

Der Bellman-Ford-Moore-Algorithmus

- Die BFM-Variante prüft in jeder Runde nur diejenigen Kanten (u, v) auf Relaxation, für die $g(u)$ in der vorigen Runde erniedrigt wurde.
- Dies führt auf eine deutliche Verbesserung der durchschnittlichen Laufzeit.
- Wurde nämlich $g(u)$ in der $(i - 1)$ -ten Runde nicht verringert, dann steht in der i -ten Runde sicher keine Relaxation der Kante (u, v) an.
- Es liegt nahe, die in der nächsten Runde zu prüfenden Knoten u in einer Schlange Q zu speichern.
- Dabei kann mit u auch die aktuelle Rundenzahl i in Q gespeichert werden.
- In Runde 0 wird der Startknoten s in Q eingefügt.
- Können in Runde n immer noch Kanten relaxiert werden, so bricht der Algorithmus mit der Fehlermeldung ab, dass negative Kreise existieren.
- Da die BFM-Variante die Kanten in derselben Reihenfolge relaxiert wie der BF-Algorithmus, führt sie auf dasselbe Ergebnis.

Der Bellman-Ford-Moore-Algorithmus

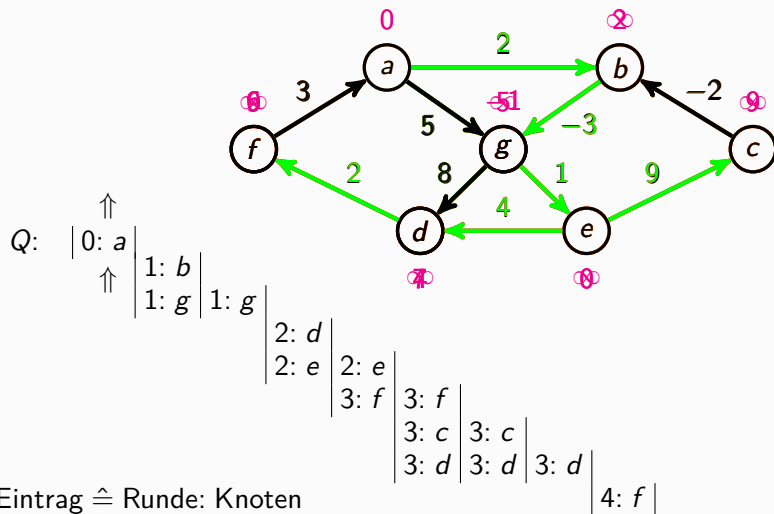
Algorithmus BFM(V, E, ℓ, s)

```
1  for all  $v \in V$  do
2       $g(v) := \infty$ ,  $\text{parent}(v) := \text{nil}$ ,  $\text{active}(v) := \text{false}$ 
3   $g(s) := 0$ ,  $\text{Init}(Q)$ ,  $\text{Enqueue}(Q, (0, s))$ ,  $\text{active}(s) := \text{true}$ 
4  while  $(i, u) := \text{Dequeue}(Q) \neq \text{nil}$  and  $i < n$  do
5       $\text{active}(u) := \text{false}$ 
6      for all  $v \in N^+(u)$  do
7          if  $g(u) + \ell(u, v) < g(v)$  then
8               $g(v) := g(u) + \ell(u, v)$ 
9               $\text{parent}(v) := u$ 
10         if  $\text{active}(v) = \text{false}$  then
11              $\text{Enqueue}(Q, (i + 1, v))$ 
12              $\text{active}(v) := \text{true}$ 
13  if  $i = n$  then
14      error(es gibt einen negativen Kreis)
```

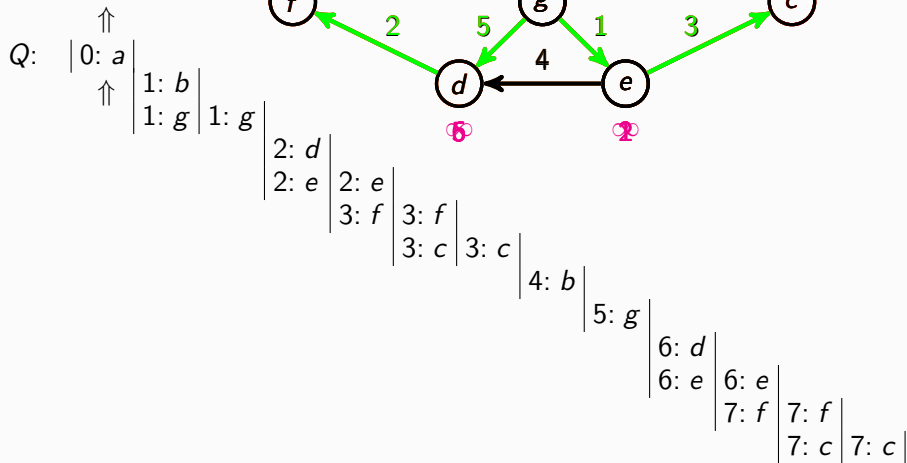
Der Bellman-Ford-Moore-Algorithmus

Beispiel

Betrachte folgenden kantenbewerteten Digraphen mit dem Startknoten a :



Der Bellman-Ford-Moore-Algorithmus



Berechnung von kürzesten Wegen zwischen allen Knoten

- Sei $G = (V, E)$ ein endlicher Digraph mit $V = \{1, \dots, n\}$.
- Weiter sei ℓ eine Funktion, die jeder Kante $(i, j) \in E$ einen Kostenwert $\ell(i, j)$ zuweist.
- Für einen Weg $W = (v_1, v_2, \dots, v_{k-1}, v_k)$ heißen $v_2, \dots, v_{k-1}$ **innere Knoten** von W .
- Bezeichne $d_k(i, j)$ die Länge eines kürzesten Weges von i nach j , der
 - mindestens eine Kante durchläuft und
 - dessen inneren Knoten alle aus der Menge $\{1, \dots, k\}$ stammen.

- Dann gilt

$$d_0(i, j) = \begin{cases} \ell(i, j), & (i, j) \in E, \\ \infty, & \text{sonst.} \end{cases}$$

und

$$d_k(i, j) = \min \{d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j)\}.$$

- Der Algorithmus von Floyd-Warshall berechnet kürzeste Wege zwischen allen Knoten, falls keine negativen Kreise existieren.

Der Floyd-Warshall-Algorithmus

Algorithmus Floyd-Warshall(V, E, ℓ)

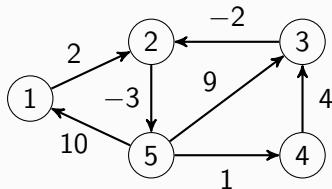
```
1  for  $i := 1$  to  $n$  do
2    for  $j := 1$  to  $n$  do
3      if  $(i, j) \in E$  then  $d_0(i, j) := \ell(i, j)$  else  $d_0(i, j) := \infty$ 
4  for  $k := 1$  to  $n$  do
5    for  $i := 1$  to  $n$  do
6      for  $j := 1$  to  $n$  do
7         $d_k(i, j) = \min \{d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j)\}$ 
```

- Die Laufzeit ist offenbar $\mathcal{O}(n^3)$.
- Da die d_k -Werte nur von den d_{k-1} -Werten abhängen, ist der Speicherplatzbedarf $\mathcal{O}(n^2)$.
- Die Existenz negativer Kreise lässt sich daran erkennen, dass mindestens ein Diagonalelement $d_k(i, i)$ einen negativen Wert erhält.

Der Floyd-Warshall-Algorithmus

Beispiel

Betrachte folgenden kantenbewerteten Digraphen:



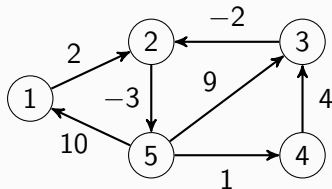
d_0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	∞	9	1	∞

d_1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10		9	1	∞

Der Floyd-Warshall-Algorithmus

Beispiel

Betrachte folgenden kantenbewerteten Digraphen:



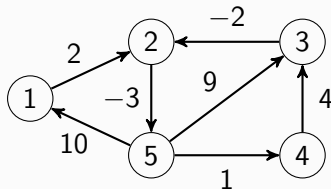
d ₁	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	12	9	1	∞

d ₂	1	2	3	4	5
1	∞	2	∞	∞	
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	
4	∞	∞	4	∞	∞
5	10	12	9	1	

Der Floyd-Warshall-Algorithmus

Beispiel

Betrachte folgenden kantenbewerteten Digraphen:



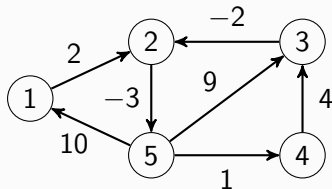
d ₂	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	∞	4	∞	∞
5	10	12	9	1	9

d ₃	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞		4	∞	
5	10		9	1	

Der Floyd-Warshall-Algorithmus

Beispiel

Betrachte folgenden kantenbewerteten Digraphen:



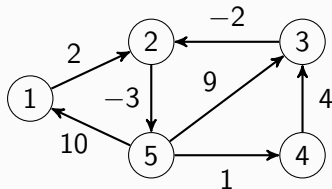
d ₃	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	7	9	1	4

d ₄	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10			1	

Der Floyd-Warshall-Algorithmus

Beispiel

Betrachte folgenden kantenbewerteten Digraphen:



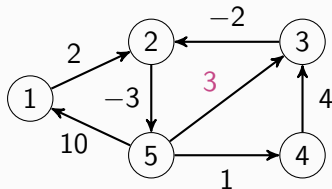
d_4	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	3	5	1	0

d_5	1	2	3	4	5
1					-1
2					-3
3					-5
4					-1
5	10	3	5	1	0

Der Floyd-Warshall-Algorithmus

Beispiel

Als nächstes betrachten wir folgenden Digraphen:



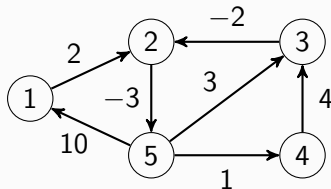
d_0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	∞	3	1	∞

d_1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10		3	1	∞

Der Floyd-Warshall-Algorithmus

Beispiel

Als nächstes betrachten wir folgenden Digraphen:



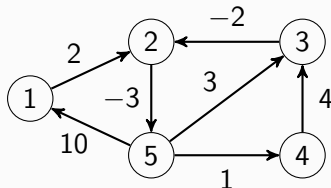
d ₁	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	12	3	1	∞

d ₂	1	2	3	4	5
1	∞	2	∞	∞	
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	
4	∞	∞	4	∞	∞
5	10	12	3	1	

Der Floyd-Warshall-Algorithmus

Beispiel

Als nächstes betrachten wir folgenden Digraphen:



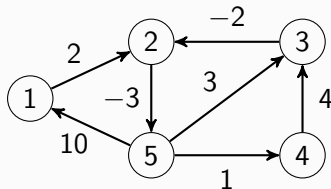
d ₂	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	∞	4	∞	∞
5	10	12	3	1	9

d ₃	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞		4	∞	
5	10		3	1	

Der Floyd-Warshall-Algorithmus

Beispiel

Als nächstes betrachten wir folgenden Digraphen:



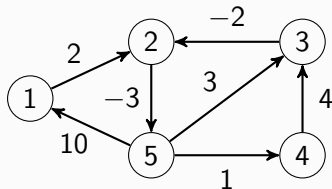
d ₃	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	1	3	1	-2

d ₄	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10			1	

Der Floyd-Warshall-Algorithmus

Beispiel

Als nächstes betrachten wir folgenden Digraphen:



d_4	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	1	3	1	-2

d_5	1	2	3	4	5
1					
2					
3					
4					
5					

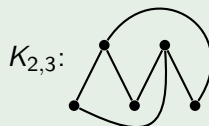
Planare Graphen

Definition

- Ein Graph G heißt **planar**, wenn er so in die Ebene einbettbar ist, dass sich zwei verschiedene Kanten höchstens in ihren Endpunkten berühren.
- Dabei werden die Knoten von G als Punkte und die Kanten von G als Verbindungslinien zwischen den zugehörigen Endpunkten dargestellt.
- Ein **ebener** Graph ist ein in die Ebene eingebetteter Graph.

Beispiel

Wie die folgenden Einbettungen von K_4 und $K_{2,3}$ in die Ebene zeigen, sind K_4 und $K_{2,3}$ planar.



Frage:

Sind auch K_5 und $K_{3,3}$ planar?

Planare Graphen

Definition

- Durch die Kanten eines ebenen Graphen wird die Ebene in so genannte **Gebiete** unterteilt.
- Nur eines dieser Gebiete ist unbeschränkt und dieses wird als **äußeres Gebiet** bezeichnet.
- Die Anzahl der Gebiete von G bezeichnen wir mit $r(G)$ oder kurz mit r .
- Der **Rand** eines Gebiets g ist die Menge aller Kanten, die an g grenzen.
- Die Anzahl der an ein Gebiet g grenzenden Kanten bezeichnen wir mit $d(g)$, wobei von g eingeschlossene Kanten doppelt gezählt werden.
- Die Gesamtzahl aller Inzidenzen von Gebieten und Kanten bezeichnen wir mit $i(G)$. Da hierbei jede Kante mit 2 Inzidenzen zu Buche schlägt, gilt

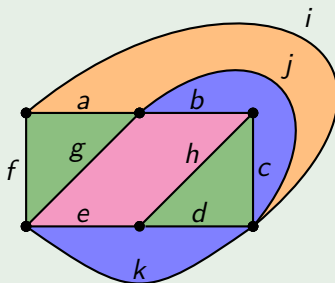
$$\sum_g d(g) = i(G) = 2m(G).$$

- Ein ebener Graph wird durch das Tripel $G = (V, E, R)$ beschrieben, wobei R aus den Rändern aller Gebiete von G besteht.

Planare Graphen

Beispiel

Der ebene Graph



hat die Darstellung $G = (V, E, R)$ mit $E = \{a, b, c, d, e, f, g, h, i, j, k\}$ und $R = \{\{a, f, g\}, \{a, i, j\}, \{b, h, e, g\}, \{b, c, j\}, \{c, d, h\}, \{e, d, k\}, \{f, i, k\}\}$.

Eulersche Polyederformel

Satz (Polyederformel von Euler, 1750)

Für einen zusammenhängenden ebenen Graphen $G = (V, E, R)$ gilt

$$n(G) - m(G) + r(G) = 2. \quad (*)$$

Beweis.

Wir führen den Beweis durch Induktion über die Kantenzahl $m(G) = m$.

$m = 0$: Da G zusammenhängend ist, folgt $n = r = 1$, d.h. $(*)$ ist erfüllt.

$m - 1 \rightsquigarrow m$: Sei G ein zusammenhängender ebener Graph mit m Kanten.

Falls G ein Baum ist, entfernen wir ein Blatt und erhalten einen zush. ebenen Graphen G' mit $n - 1$ Knoten, $m - 1$ Kanten und r Gebieten. Nach IV folgt $(n - 1) - (m - 1) + r = 2$, d.h. $(*)$ ist erfüllt.

Falls G kein Baum ist, entfernen wir eine Kante auf einem Kreis und erhalten einen zush. ebenen Graphen G' mit n Knoten, $m - 1$ Kanten und $r - 1$ Gebieten. Nach IV folgt $n - (m - 1) + (r - 1) = 2$, d.h. $(*)$ ist erfüllt. □

Planare Graphen haben nur wenige Kanten

Korollar

Für jeden planaren Graphen mit $n \geq 3$ Knoten gilt $m \leq 3n - 6$.

Beweis.

- O.B.d.A. sei G zusammenhängend.
- Da $n \geq 3$ ist, ist jedes Gebiet g von $d(g) \geq 3$ Kanten umgeben.
- Daher ist $2m = i = \sum_g d(g) \geq 3r$ bzw. $r \leq 2m/3$.
- Eulers Formel liefert $m = n + r - 2 \leq n + 2m/3 - 2$, was $(1 - 2/3)m \leq n - 2$ und somit $m \leq 3n - 6$ impliziert. □

Korollar

K_5 ist nicht planar.

Beweis.

Wegen $n = 5$, also $3n - 6 = 9$, und wegen $m = \binom{5}{2} = 10$ gilt $m \not\leq 3n - 6$. □

Dreiecksfreie planare Graphen haben noch weniger Kanten

Korollar

Für dreiecksfreie planare Graphen mit $n \geq 3$ Knoten gilt $m \leq 2n - 4$.

Beweis.

- Wir betrachten eine beliebige planare Einbettung von G , wobei wir wieder o.B.d.A. annehmen, dass G zusammenhängend ist.
- Da G dreiecksfrei ist, ist jedes Gebiet von $d(g) \geq 4$ Kanten umgeben.
- Daher ist $2m = i = \sum_g d(g) \geq 4r$ bzw. $r \leq m/2$.
- Eulers Formel liefert $m = n + r - 2 \leq n + m/2 - 2$ bzw. $m/2 \leq n - 2$, was $m \leq 2n - 4$ impliziert. $\square$

Korollar

$K_{3,3}$ ist nicht planar.

Beweis.

Wegen $n = 6$, also $2n - 4 = 8$, und wegen $m = 3 \cdot 3 = 9$ gilt $m \not\leq 2n - 4$. $\square$

Planare Graphen haben kleinen Minimalgrad

Als weitere interessante Folgerung aus der Polyederformel können wir zeigen, dass jeder planare Graph einen Knoten v vom Grad $\deg(v) \leq 5$ hat.

Lemma

Jeder planare Graph hat einen Minimalgrad $\delta(G) \leq 5$.

Beweis.

- Für $n \leq 6$ ist die Behauptung klar.
- Für $n > 6$ impliziert die Annahme $\delta(G) \geq 6$ die Ungleichung

$$m = \frac{1}{2} \sum_{u \in V} \deg(u) \geq \frac{1}{2} \sum_{u \in V} 6 = 3n.$$

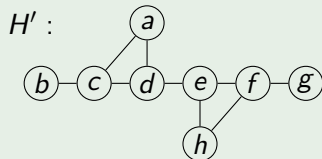
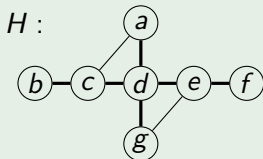
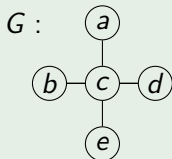
was im Widerspruch zu $m \leq 3n - 6$ steht.



Überbrückungen und Unterteilungen

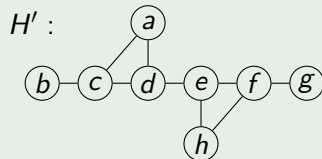
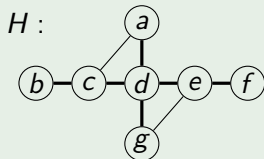
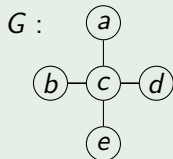
Definition

- Sei v ein Knoten vom Grad 2 in G mit den beiden Nachbarn u und w , so entsteht der Graph $G' := (G - v) \cup \{u, w\}$ durch **Überbrückung** von v .
- H heißt **Unterteilung** von G , wenn G durch wiederholte Anwendung der Operation Überbrückung aus einer isomorphen Kopie von H entsteht.



- Ist H eine Unterteilung von G ?
- Ist ein Teilgraph von H eine Unterteilung von G ?

Überbrückungen und Unterteilungen



- Ist H eine Unterteilung von G ?
- Ist ein Teilgraph von H eine Unterteilung von G ?
- Ist H' eine Unterteilung von G ?
- Ist ein Teilgraph von H' eine Unterteilung von G ?

Die Minorenrelation

Definition

Sei $G = (V, E)$ ein Graph und seien $u, v \in V$.

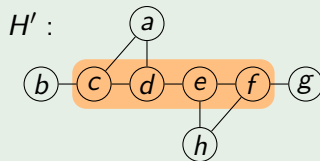
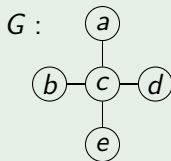
- Durch **Fusion** von u und v entsteht dann der Graph

$$G_{uv} = (V - \{v\}, E') \text{ mit}$$

$$E' = \{e \in E \mid v \notin e\} \cup \{\{u, v'\} \mid \{v, v'\} \in E - \{u, v\}\}.$$

- Ist $e = \{u, v\}$ eine Kante von G (also $e \in E$), so sagen wir auch, G_{uv} entsteht aus G durch **Kontraktion** der Kante e .
- G heißt **Minor** von H , wenn sich ein zu G isomorpher Graph durch wiederholte Anwendung folgender Operationen aus H gewinnen lässt:
 - Entfernen einer Kante oder eines Knotens und
 - Kontraktion einer Kante.
- H heißt **G-frei**, falls G kein Minor von H ist.

Die Minorenrelation



Ist G ein Minor von H' ?

Die Sätze von Kuratowski und Wagner

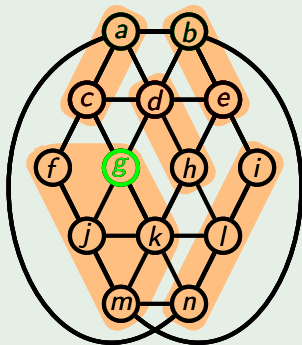
Kuratowski konnte 1930 beweisen, dass in jedem nichtplanaren Graphen G eine Unterteilung des $K_{3,3}$ oder des K_5 als Teilgraph enthalten sein muss. Einige Jahre später beobachtete Wagner, dass dann auch der $K_{3,3}$ oder der K_5 ein Minor von G sein muss.

Satz (Kuratowski 1930, Wagner 1937)

Sei G ein Graph. Dann sind folgende Aussagen äquivalent:

- G ist planar.
- Kein Teilgraph von G ist eine Unterteilung des $K_{3,3}$ oder K_5 .
- G ist $K_{3,3}$ -frei und K_5 -frei.

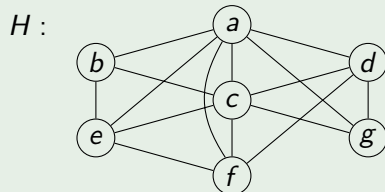
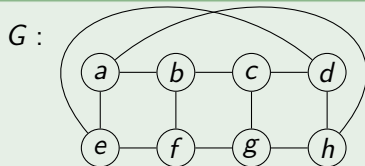
Beispiel zu den Sätzen von Kuratowski und Wagner



- Durch Entfernen der dünnen Kanten entsteht eine Unterteilung des K_5 :
Überbrücke die dünn umrandeten Knoten.
- Alternativ können wir den K_5 durch Fusion gewinnen.
- Oder eine Unterteilung des $K_{3,3}$.
- Oder eine weitere Unterteilung des K_5 .

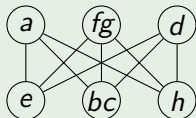
Aufgabe 9 der Probeklausur

Welche der beiden folgenden Graphen sind planar und welche nicht. Begründen Sie.

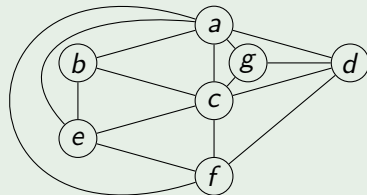


Der Graph G ist nicht planar, weil er den $K_{3,3}$ als Minor enthält:
Verschmelze die Knoten b und c sowie die Knoten f und g .

G' :



H :



Färbung von Graphen

Definition

- Eine Abbildung $c: V \rightarrow \mathbb{N}$ heißt **Färbung** von G , wenn $c(u) \neq c(v)$ für alle $\{u, v\} \in E$ gilt. G heißt **k -färbbar**, falls eine Färbung $c: V \rightarrow \{1, \dots, k\}$ existiert.
- Die **chromatische Zahl** ist

$$\chi(G) = \min\{k \in \mathbb{N} \mid G \text{ ist } k\text{-färbbar}\}.$$

- Ein Graph heißt **bipartit**, wenn $\chi(G) \leq 2$ ist.

Beispiel

$$\chi(E_n) = 1, \chi(K_{n,m}) = 2, \chi(K_n) = n,$$

$$\chi(C_n) = \begin{cases} 2, & n \text{ gerade} \\ 3, & \text{sonst.} \end{cases}$$

Färbung von planaren Graphen

Frage:

Wie viele Farben werden höchstens benötigt, um eine Landkarte so zu färben, dass aneinander grenzende Länder unterschiedliche Farben erhalten.

- Offensichtlich lässt sich eine Landkarte in einen planaren Graphen transformieren, indem man für jedes Land einen Knoten zeichnet und benachbarte Länder durch eine Kante verbindet. Länder, die sich nur in einem Punkt berühren, gelten nicht als benachbart.
- Bereits im 19. Jahrhundert wurde vermutet, dass 4 Farben ausreichen.
- Diese Vermutung wurde 1878 von Kempe „bewiesen“ und erst 1890 entdeckte Heawood einen Fehler in Kempes „Beweis“.
- Übrig blieb der **5-Farben-Satz**.
- Der **4-Farben-Satz** wurde erst 1976 von Appel und Haken bewiesen.
- Hierbei handelt es sich jedoch nicht um einen Beweis im klassischen Sinne, da zur Überprüfung der vielen auftretenden Spezialfälle Computer benötigt werden.

Der 4-Farben-Satz

4-Farben-Satz (Appel, Haken 1976)

Jeder planare Graph ist 4-färbbar.

- Der Beweis des 4-Farben-Satzes von Appel und Haken liefert einen effizienten 4-Färbungsalgorithmus für planare Graphen mit einer Laufzeit von $\mathcal{O}(n^4)$.
- In 1997 fanden Robertson, Sanders, Seymour und Thomas eine Möglichkeit, den Beweis von Appel und Haken zu vereinfachen.
- Diese Vereinfachung führt zu einem deutlich schnelleren Algorithmus zur 4-Färbung von planaren Graphen mit quadratischer Laufzeit $\mathcal{O}(n^2)$.
- Ohne Computer-Unterstützung ist aber auch dieser Beweis nicht verifizierbar.

Beweis des 5-Farben-Satzes

5-Farben-Satz (Kempe 1878, Heawood 1890)

Jeder planare Graph ist 5-färbbar.

Beweis des 5-Farben-Satzes.

Wir führen den Beweis durch Induktion über die Knotenzahl n von G .

$n = 1$: Klar.

$n - 1 \rightsquigarrow n$: Da G planar ist, hat G einen Knoten u vom Grad $\deg(u) \leq 5$.

Zunächst entfernen wir u aus G .

Falls u fünf Nachbarn hat, sind mindestens zwei von ihnen nicht durch eine Kante verbunden und wir verschmelzen zwei solche Nachbarn v und w zu einem Knoten z .

Der resultierende Graph G' ist planar und hat $n' \leq n - 1$ Knoten. Daher existiert nach IV eine 5-Färbung c für G' .

Trennen wir nun die zu z verschmolzenen Nachbarn v und w wieder, so ist c immer noch eine 5-Färbung, die die Nachbarn von u mit höchstens 4 Farben färbt.

Daher können wir u mit der verbliebenen Farbe färben.



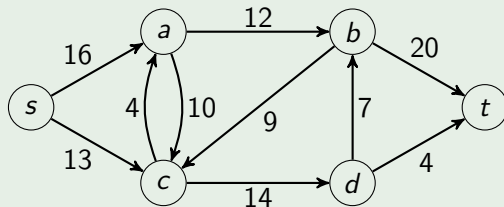
Flüsse in Netzwerken

Definition

- Ein **Netzwerk** $N = (V, E, s, t, c)$ besteht aus einem gerichteten Graphen $G = (V, E)$ mit einer **Quelle** $s \in V$ und einer **Senke** $t \in V$ sowie einer **Kapazitätsfunktion** $c : V \times V \rightarrow \mathbb{N}$.
- Allen Kanten $(u, v) \in E$ müssen positive Kapazität $c(u, v) > 0$ und alle Nichtkanten $(u, v) \notin E$ müssen die Kapazität $c(u, v) = 0$ haben.

Beispiel

Ein Netzwerk N :



Flüsse in Netzwerken

Definition

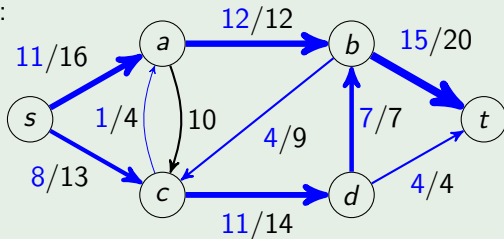
Ein **Fluss** in N ist eine Funktion $f : V \times V \rightarrow \mathbb{Z}$ mit

- $f(u, v) \leq c(u, v)$, „Kapazitätsbedingung“
- $f(u, v) = -f(v, u)$, „Symmetriebedingung“
- Für alle $u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0$. „Kontinuitätsbedingung“

Die **Größe** von f ist $|f| = \sum_{v \in V} f(s, v)$.

Beispiel

Ein Fluss f in N :

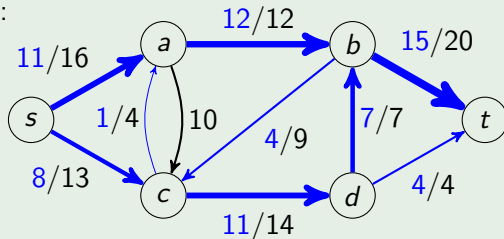


Die Symmetriebedingung impliziert, dass $f(u, u) = 0$ für alle $u \in V$ ist. ◀

Flüsse in Netzwerken

Beispiel

Ein Fluss f in N :



Die Symmetriebedingung impliziert, dass $f(u, u) = 0$ für alle $u \in V$ ist. $\triangleleft$

Wie lässt sich feststellen, ob ein geg. Fluss f vergrößert werden kann?

- Falls f der konstante Nullfluss $f = 0$ ist, genügt es, in $G = (V, E)$ einen Pfad von s nach t zu finden.
- Andernfalls können wir zu N und f ein Netzwerk N_f konstruieren, so dass f genau dann vergrößert werden kann, wenn sich in N_f der Nullfluss vergrößern lässt.

Flüsse in Netzwerken

Definition

- Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei f ein Fluss in N .
- Das zugeordnete **Restnetzwerk** ist $N_f = (V, E_f, s, t, c_f)$, wobei

$$c_f(u, v) = c(u, v) - f(u, v)$$

und

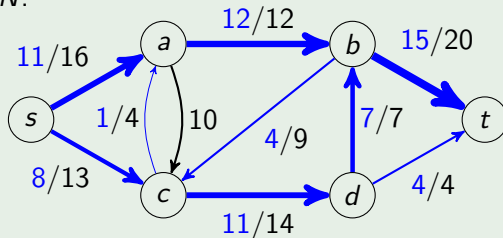
$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

ist.

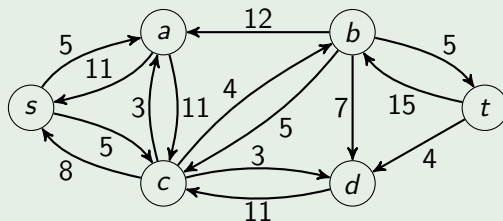
Flüsse in Netzwerken

Beispiel

- Ein Fluss f in N :



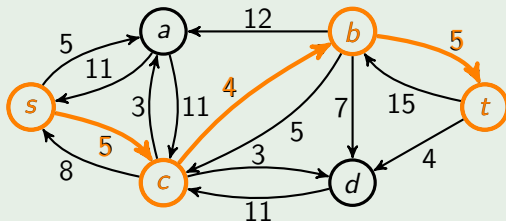
- Das zu N und f gehörige Restnetzwerk N_f :



Flüsse in Netzwerken

Beispiel

- Das zu N und f gehörige Restnetzwerk N_f :



- In N_f gibt es einen Pfad P von s nach t :

$$P = (s, c, b, t).$$

Flüsse in Netzwerken

Definition

- Sei $N_f = (V, E_f, s, t, c_f)$ ein Restnetzwerk.
- Dann heißt jeder s - t -Pfad P in (V, E_f) **Zunahmepfad** in N_f .
- Die **Kapazität von P** in N_f ist

$$c_f(P) = \min\{c_f(u, v) \mid (u, v) \text{ liegt auf } P\}$$

und der **zu P gehörige Fluss in N_f** ist

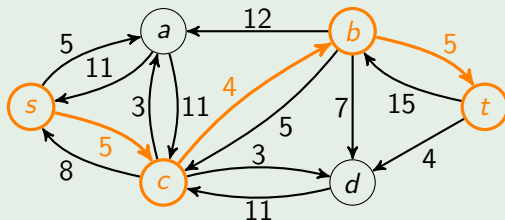
$$f_P(u, v) = \begin{cases} c_f(P), & (u, v) \text{ liegt auf } P, \\ -c_f(P), & (v, u) \text{ liegt auf } P, \\ 0, & \text{sonst.} \end{cases}$$

$P = (u_0, \dots, u_k)$ ist also genau dann ein Zunahmepfad in N_f , falls

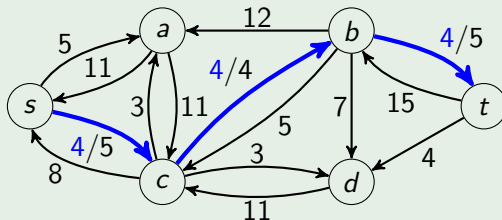
- $u_0 = s$ und $u_k = t$ ist,
- die Knoten $u_0, \dots, u_k$ paarweise verschieden sind
- und $c_f(u_i, u_{i+1}) > 0$ für $i = 0, \dots, k - 1$ ist.

Flüsse in Netzwerken

Beispiel



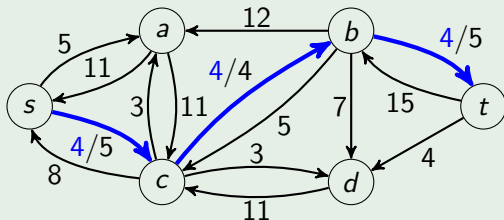
- Der Pfad $P = (s, c, b, t)$ hat die Kapazität $c_f(P) = \min\{5, 4, 5\} = 4$.
- Der zugehörige Fluss f_P in N_f ist



Flüsse in Netzwerken

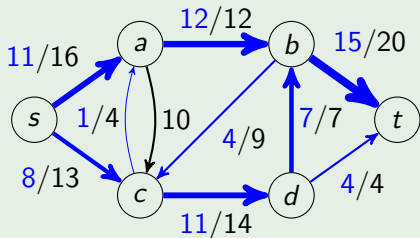
Beispiel

Fluss f_P in N_f :

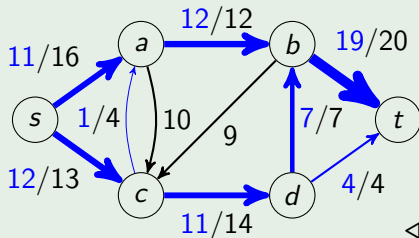


Addieren wir f_P auf den Fluss f in N , so erhalten wir den Fluss $f + f_P$:

Fluss f :



Fluss $f + f_P$:



Der Ford-Fulkerson-Algorithmus

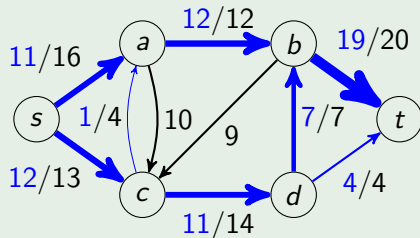
Algorithmus Ford-Fulkerson(V, E, s, t, c)

```
1  for all  $(u, v) \in V \times V$  do
2     $f(u, v) := 0$ 
3  while es gibt einen Zunahmepfad  $P$  in  $N_f$  do
4     $f := f + f_P$ 
```

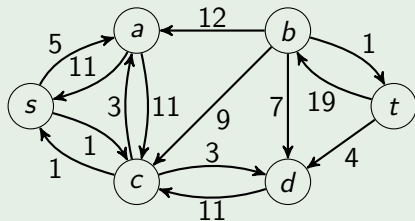
Beispiel

- Für den neuen Fluss f erhalten wir nun folgendes Restnetzwerk N_f :

Fluss f :



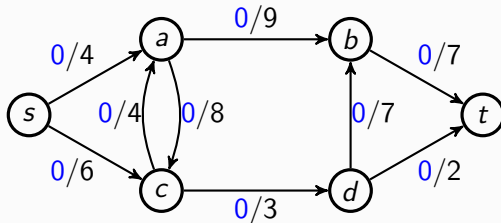
N_f :



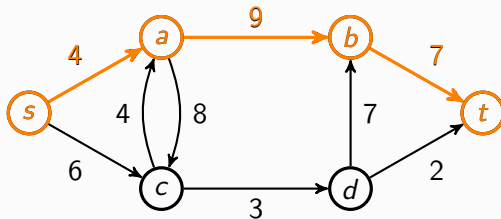
- In diesem existiert kein Zunahmepfad mehr.

Ford-Fulkerson-Algorithmus

- Betrachte folgendes Netzwerk N . Wir beginnen mit dem Fluss $f_0 = 0$:



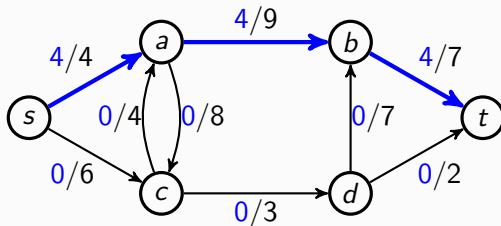
- Der Fluss f_0 führt auf das Restnetzwerk $N_{f_0} = N$:



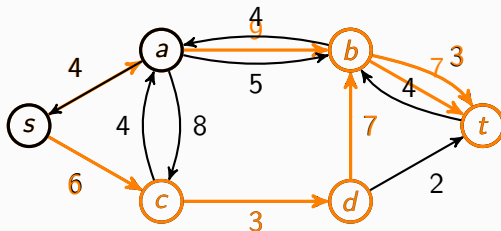
Zunahmepfad

Ford-Fulkerson-Algorithmus

- Der Zunahmepfad P_1 führt auf den Fluss $f_1 = f_0 + f_{P_1}$:



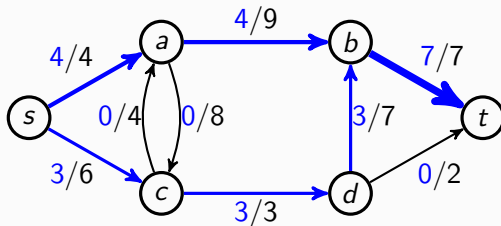
- Der Fluss f_0 führt auf das Restnetzwerk $N_{f_0} = N$:



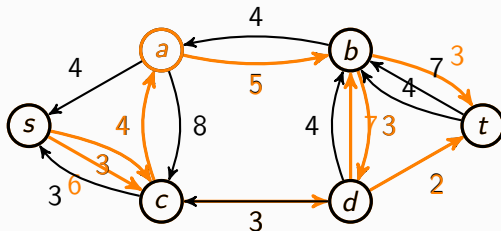
Zunahmepfad $P_1 = (s, a, b, t)$.

Ford-Fulkerson-Algorithmus

- Der Zunahmepfad P_2 führt auf den Fluss $f_2 = f_1 + f_{P_2}$:



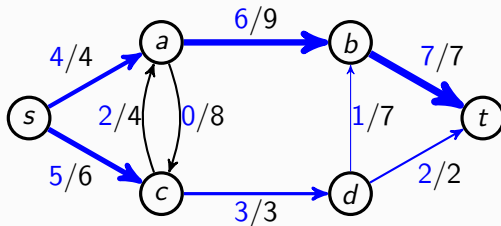
- Der Fluss f_1 führt auf das Restnetzwerk N_{f_1} :



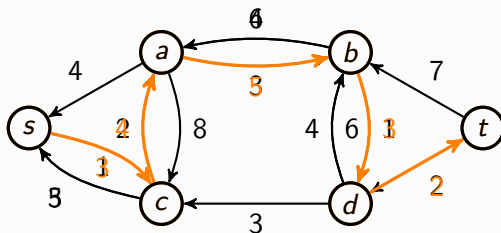
Zunahmepfad $P_2 = (s, c, d, b, t)$.

Ford-Fulkerson-Algorithmus

- Der Zunahmepfad P_3 führt auf den Fluss $f_3 = f_2 + f_{P_3}$:



- Der Fluss f_2 führt auf das Restnetzwerk N_{f_2} :



Zunahmepfad $P_3 = (s, c, a, b, d, t)$.

Der Ford-Fulkerson-Algorithmus

Algorithmus Ford-Fulkerson(V, E, s, t, c)

```
1  for all  $(u, v) \in V \times V$  do
2     $f(u, v) := 0$ 
3  while es gibt einen Zunahmepfad  $P$  in  $N_f$  do
4     $f := f + f_P$ 
```

Berechnet Ford-Fulkerson tatsächlich einen Maximalfluss?

Hierzu müssen wir zeigen, dass f ein Maximalfluss ist, falls es im Restnetzwerk N_f keinen Zunahmepfad mehr gibt.

Korrektheit des Ford-Fulkerson-Algorithmus

Definition

- Sei $N = (V, E, s, t, c)$ ein Netzwerk und seien $S, T \subseteq V$ mit $S \cap T = \emptyset$ und $S \cup T = V$.
- Dann heißt (S, T) ein **Schnitt** durch N , falls $s \in S$ und $t \in T$ ist.
- Die **Kapazität** eines Schnittes (S, T) ist

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

- Ist f ein Fluss in N , so heißt

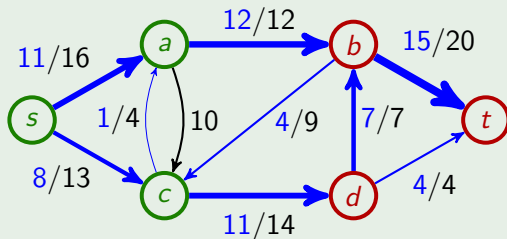
$$f(S, T) = \sum_{u \in S, v \in T} f(u, v)$$

der **Fluss durch den Schnitt** (S, T) .

Flüsse in Netzwerken

Beispiel

- Betrachte den Schnitt (S, T) mit $S = \{s, a, c\}$ und $T = \{b, d, t\}$ in folgendem Netzwerk N mit dem Fluss f :



- Dieser Schnitt hat die Kapazität

$$c(S, T) = c(a, b) + c(c, d) = 12 + 14 = 26$$

und der Fluss durch diesen Schnitt ist

$$f(S, T) = f(a, b) + f(c, b) + f(c, d) = 12 - 4 + 11 = 19.$$

Korrektheit des Ford-Fulkerson-Algorithmus

Lemma

Für jeden Fluss f und jeden Schnitt (S, T) gilt

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

Beweis.

- Wir zeigen die Gleichheit $|f| = f(S, T)$ durch Induktion über $k = \|S\|$.

$k = 1$: In diesem Fall ist $S = \{s\}$ und $T = V - \{s\}$. Daher folgt

$$|f| = \sum_{v \in V - \{s\}} f(s, v) = \sum_{u \in S, v \in T} f(u, v) = f(S, T).$$

Korrektheit des Ford-Fulkerson-Algorithmus

Beweis (Fortsetzung).

$k - 1 \rightsquigarrow k$: Sei (S, T) ein Schnitt mit $\|S\| = k > 1$ und sei $w \in S - \{s\}$.

Betrachte den Schnitt (S', T') mit $S' = S - \{w\}$ und $T' = T \cup \{w\}$.

Dann gilt

$$f(S, T) = \sum_{u \in S, v \in T} f(u, v) = \sum_{u \in S', v \in T} f(u, v) + \sum_{v \in T} f(w, v)$$

und

$$f(S', T') = \sum_{u \in S', v \in T'} f(u, v) = \sum_{u \in S', v \in T} f(u, v) + \sum_{u \in S'} f(u, w).$$

Wegen $f(w, w) = 0$ ist $\sum_{u \in S'} f(u, w) = \sum_{u \in S} f(u, w)$ und daher folgt

$$f(S, T) - f(S', T') = \sum_{v \in T} f(w, v) - \sum_{u \in S} f(u, w) = \sum_{v \in V} f(w, v) = 0.$$

Nach Induktionsvoraussetzung ist somit $f(S, T) = f(S', T') = |f|$.

Korrektheit des Ford-Fulkerson-Algorithmus

Lemma

Für jeden Fluss f und jeden Schnitt (S, T) gilt

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

Beweis (Ende).

- Wir zeigen $|f| = f(S, T)$ durch Induktion über $k = \|S\|$. ✓
- Schließlich folgt wegen $f(u, v) \leq c(u, v)$ die Ungleichung

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

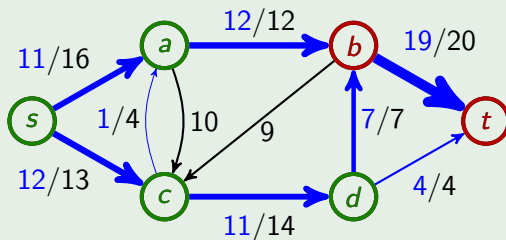


Korrektheit des Ford-Fulkerson-Algorithmus

Min-Cut-Max-Flow-Theorem

Sei f ein Fluss in $N = (V, E, s, t, c)$. Dann sind folg. Aussagen äquivalent:

- 1 f ist maximal.
- 2 In N_f existiert kein Zunahmepfad.
- 3 Es gibt einen Schnitt (S, T) mit $c(S, T) = |f|$.



Für den Schnitt (S, T) mit $S = \{s, a, c, d\}$ und $T = \{b, t\}$ gilt

$$c(S, T) = c(a, b) + c(d, b) + c(d, t) = f(a, b) + f(d, b) + f(c, b) + f(d, t) = |f|.$$

Korrektheit des Ford-Fulkerson-Algorithmus

Min-Cut-Max-Flow-Theorem

Sei f ein Fluss in $N = (V, E, s, t, c)$. Dann sind folg. Aussagen äquivalent:

- 1 f ist maximal.
- 2 In N_f existiert kein Zunahmepfad.
- 3 Es gibt einen Schnitt (S, T) mit $c(S, T) = |f|$.

Beweis.

- Die Implikation von 1 nach 2 ist klar, da die Existenz eines Zunahmepfades zu einer Vergrößerung von f führt.

Korrektheit des Ford-Fulkerson-Algorithmus

Min-Cut-Max-Flow-Theorem

Sei f ein Fluss in $N = (V, E, s, t, c)$. Dann sind folg. Aussagen äquivalent:

- 1 f ist maximal.
- 2 In N_f existiert kein Zunahmepfad.
- 3 Es gibt einen Schnitt (S, T) mit $c(S, T) = |f|$.

Beweis (Fortsetzung).

- Für die Implikation von 2 nach 3 betrachten wir den Schnitt (S, T) mit $S = \{u \in V \mid u \text{ ist in } N_f \text{ von } s \text{ aus erreichbar}\}$ und $T = V - S$.
- Da in N_f kein Zunahmepfad existiert, gilt dann

$$s \in S, t \in T \text{ und } c_f(u, v) = 0 \text{ für alle } u \in S \text{ und } v \in T.$$

- Wegen $c_f(u, v) = c(u, v) - f(u, v)$ folgt somit

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

Korrektheit des Ford-Fulkerson-Algorithmus

Min-Cut-Max-Flow-Theorem

Sei f ein Fluss in $N = (V, E, s, t, c)$. Dann sind folg. Aussagen äquivalent:

- 1 f ist maximal.
- 2 In N_f existiert kein Zunahmepfad.
- 3 Es gibt einen Schnitt (S, T) mit $c(S, T) = |f|$.

Beweis (Schluss).

- Die Implikation von 3 nach 1 ist wieder klar, da im Fall $c(S, T) = |f|$ für jeden Fluss f' gilt:

$$|f'| = f'(S, T) \leq c(S, T) = |f|.$$



Laufzeit des Ford-Fulkerson-Algorithmus

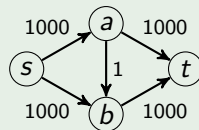
Bemerkung

- Sei $c_0 = \sum_{v \in V} c(s, v)$ die Kapazität des Schnittes (S, T) mit $S = \{s\}$.
- Dann durchläuft der Ford-Fulkerson-Algorithmus die while-Schleife höchstens c_0 -mal.
- Bei jedem Durchlauf ist zuerst das Restnetzwerk N_f und danach ein Zunahmepfad in N_f zu berechnen.
- Die Berechnung des Zunahmepfads P kann durch Breitensuche in Zeit $\mathcal{O}(n + m)$ erfolgen.
- Da sich das Restnetzwerk nur entlang von P ändert, kann es in Zeit $\mathcal{O}(n)$ aktualisiert werden.
- Jeder Durchlauf benötigt also Zeit $\mathcal{O}(n + m)$, was auf eine Gesamtlaufzeit von $\mathcal{O}(c_0(n + m))$ führt.
- Da der Wert von c_0 jedoch exponentiell in der Länge der Eingabe (also der Beschreibung des Netzwerkes N) sein kann, ergibt dies keine polynomielle Zeitschranke.

Laufzeit des Ford-Fulkerson-Algorithmus

Beispiel

- Bei nebenstehendem Netzwerk benötigt Ford-Fulkerson zur Bestimmung des Maximalflusses abhängig von der Wahl der Zunahmepfade ? Schleifendurchläufe.



- Wir beginnen mit dem Nullfluss $f = 0$, der auf das Restnetzwerk $N_f = N$ führt und wählen darin den Zunahmepfad $P = (s, a, b, t)$.

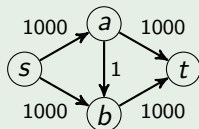
Restnetzwerk	Zunahmepfad	neuer Fluss

Restnetzwerk	Zunahmepfad	neuer Fluss

Beschleunigung des Ford-Fulkerson-Algorithmus

Beispiel

Bei nebenstehendem Netzwerk benötigt Ford-Fulkerson zur Bestimmung des Maximalflusses abhängig von der Wahl der Zunahmepfade zwischen 2 und 2000 Schleifendurchläufe.



Bemerkung

Nicht nur in diesem Beispiel lässt sich dies wie folgt vermeiden:

- Man betrachtet nur Zunahmepfade mit einer geeignet gewählten Mindestkapazität. Dies führt auf eine Laufzeit, die polynomiell in n , m und $\log c_0$ ist.
- Man bestimmt in jeder Iteration einen kürzesten Zunahmepfad. Diese Vorgehensweise wird als **Edmonds-Karp-Strategie** bezeichnet und führt auf eine Laufzeit von $\mathcal{O}(nm^2)$ (unabhängig von der Kapazitätsfunktion).
- Man bestimmt in jeder Iteration einen **blockierenden Fluss** (Algorithmus von Dinic). Dies führt auf eine Laufzeit von $\mathcal{O}(n^2m)$.