Algorithmen und Datenstrukturen
Kapitel 11
Zusammenfassung

Frank Heitmann
heitmann@informatik.uni-hamburg.de

20. Januar 2016

Was ist klausurrelevant?

Was ist klausurrelevant/Grundlage der Prüfung?

- Die Vorlesung
- Die Aufgabenzettel/Musterlösungen.

Anmerkung

Die Lektüre der jeweiligen Kapitel im [Cormen] ist bei Unklarheiten nützlich!

Wichtige Anmerkung

Bei Flüssen gilt die Definition auf den Folien! (Unterschied zwischen der alten und neuen Auflage des [Cormen].)

Allgemeines Herangehen

Generell solltet ihr zu jedem behandelten Algorithmus wissen:

- Wie sieht die **Eingabe** aus? (Was ist das Problem?)
- Wie sieht die **Ausgabe** aus? (Was ist das Ziel?)
- Wie ist die **Laufzeit**?
- Was ist die **Idee** dieses Algorithmus?
- Wie **arbeitet** der Algorithmus?

Anmerkung

Idee und Laufzeit bedingen sich. Wenn ihr die Idee kennt, habt ihr meist auch eine gute Vorstellung der Laufzeit. Bei der Arbeitsweise gilt: Je genauer desto besser. Sourcecode wird aber nicht abgefragt. Wie bei den Übungszetteln solltet ihr *die wichtigen* Algorithmen an kleinen Beispielen ausführen können.

Zur Motivation

Anmerkung zur Motivation/zum Nutzen

Die Ideen der verschiedenen Algorithmen zu kennen ist beim eigenen Algorithmenentwurf enorm hilfreich. - Mergesort o.ä. zu implementieren dürfte den wenigsten begegnen, aber ein Problem durch ein Divide & Conquer-Verfahren zu attackieren schon eher!

Aufteilung der Klausur

Grob entspricht die Aufteilung der Klausur den Anteil der Themen in der Vorlesung, d.h.:

- $\approx 25\%$ O-Notation und Rekurrenzgleichungen
- $\approx 25\%$ Sortieren und Suchen (inkl. Suchbäume, etc.)
- $\approx 30\%$ Graphen
- $\approx 20\%$ P, NP usw.

Art der Fragen

An Fragen gibt es

- $\approx 60\%$ Verfahren benutzen, Rechnen, Wissen wiedergeben z.B.
  - im binären Suchbaum löschen
  - Kruskals Algorithmus anwenden
  - $n^2 \in O(n^3)$ zeigen/widerlegen
- $\approx 30\%$ schwierige Verfahren oder spezielles Wissen abfragen, kreative Verfahren im bekannten Umfeld z.B.
  - Algorithmus entwerfen, der ähnlich zu etwas bekanntem ist
  - NP-Vollständigkeitsbeweis, der ähnlich zu etwas bekanntem ist
- $\approx 10\%$ schwere Aufgaben, Out-of-the-Box-Thinking
  - ?

ab	50	55	60	65	70	75	80	85	90	95
	4,0	3,7	3,3	3,0	2,7	2,3	2,0	1,7	1,3	1,0

Vorbemerkung

Wichtige Anmerkung

In der folgenden Zusammenfassung gehen wir **recht zügig** durch den Stoff der letzten Monate. Wenn ihr einen Teil nicht genau versteht, guckt noch einmal in das jeweilige Kapitel, guckt ggf. das Vorlesungsvideo und lest im [Cormen] nach. **Benutzt nicht nur die Folien der Zusammenfassung zum Lernen!**

O-Notation

Zusammenfassung:

O-**Notation** und Rekurrenzgleichungen

O-Notation - Definition

Definition (O-Notation (und Verwandte))

- $O(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\Omega(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $o(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\omega(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|\}$

Bemerkung

f ist dabei stets eine Funktion von $\mathbb{N}$ **nach** $\mathbb{R}$, also $f : \mathbb{N} \to \mathbb{R}$. (Wegen der Änderung des Bildbereiches stehen nun auch überall die **Betragsstriche**!) In der Literatur gibt es hier Varianten. Bei der tatsächlichen Algorithmenanalyse machen diese aber kaum einen Unterschied.

O-Notation - Variante

Satz

1. $f(n) \in O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
2. $f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
3. $f(n) \in \Omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
4. $f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
5. $f(n) \in \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, 0 < c < \infty$

Merkhilfe

Bemerkung

Eine kleine *Merkhilfe* (nicht mehr!)

- $f \in O(g) \approx f \leq g$
- $f \in \Omega(g) \approx f \geq g$
- $f \in \Theta(g) \approx f = g$
- $f \in o(g) \approx f < g$
- $f \in \omega(g) \approx f > g$

Wir sagen, dass f asymptotisch kleiner gleich, größer gleich, gleich, kleiner bzw. größer ist als g, wenn $f \in O(g), f \in \Omega(g), f \in \Theta(g), f \in o(g)$ bzw. $f \in \omega(g)$ gilt.

Wichtige Eigenschaften

Satz

1. $f \in X(g)$ *und* $g \in X(h)$ *impliziert* $f \in X(h)$ $(X \in \{O, \Omega, \Theta, o, \omega\})$
2. $f \in X(f)$ $(X \in \{O, \Omega, \Theta\})$
3. $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$

Weitere wichtige Eigenschaften

Satz

1. $g \in O(f) \Leftrightarrow f \in \Omega(g)$
2. $g \in o(f) \Leftrightarrow f \in \omega(g)$
3. $g \in \Theta(f) \Leftrightarrow g \in O(f) \cap \Omega(f)$
4. $\Omega(f) \cap O(f) = \Theta(f)$
5. $o(f) \subseteq O(f)$
6. $\omega(f) \subseteq \Omega(f)$
7. $\omega(f) \cap o(f) = \emptyset$

Noch mehr wichtige Eigenschaften

Satz

1. $f, g \in O(h) \Rightarrow f + g \in O(h)$
2. $f \in O(g), c \in R^+ \Rightarrow c \cdot f \in O(g)$
3. $f \in O(h_1), g \in O(h_2) \Rightarrow f \cdot g \in O(h_1 \cdot h_2)$

Bemerkung

In der Literatur findet man statt $f \in O(g)$ oft auch $f = O(g)$.

Wichtige Funktionen

Bemerkung

Wichtige Funktionen in der Algorithmik/Algorithmenanalyse:

- Konstante Funktionen
- Logarithmen ($\log, \ln$)
- Wurzelfunktionen ($\sqrt{n}$)
- Polynome (beachte: $\sqrt{n} = n^{1/2}$)
- Exponentialfunktionen (2^n)
- ... Kombinationen davon

O-Notation

Zusammenfassung:

O-Notation und **Rekurrenzgleichungen**

Die Türme von Hanoi - Analyse. Methode 2

Methode 2. Rekursives Einsetzen (Substitutions- oder Abwicklungsmethode)

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1)+1 & \text{, falls } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 = 4T(n-2)+2+1 \\ &= 4(2T(n-3)+1)+2+1 = 8T(n-3)+4+2+1 \\ &= \ldots \\ &= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0 \ (*) \end{aligned}$$

(*) Diese Zeile ist mit vollständiger Induktion zu zeigen!

Die Türme von Hanoi - Analyse. Methode 2

Methode 2. Rekursives Einsetzen (Substitutions- oder Abwicklungsmethode)

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1) + 1 & \text{, falls } n \geq 2 \end{cases}$$

$$T(n) \quad = \quad 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0$$

Wann bricht die Rekursion ab? Bei $k = n - 1$

Die Türme von Hanoi - Analyse. Methode 2

Methode 2. Rekursives Einsetzen (Substitutions- oder Abwicklungsmethode)

$$T(n) = \begin{cases} 1 & \text{, falls } n = 1 \\ 2T(n-1) + 1 & \text{, falls } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0 \ (*) \\ &= 2^{n-1} T(n-(n-1)) + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 \\ &= \sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2 - 1} = 2^n - 1 \in \Theta(2^n) \end{aligned}$$

(*) Da bei $T(1)$, also $n - k = 1$, die Rekursion abbricht, wird mit $k = n - 1$ weiter gerechnet.

Ein weitere Beispiel

$$T(n) = \begin{cases} c & , \text{falls } n = 1 \\ 3T(n/3) + 1/n & , \text{falls } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 3 \cdot T(\frac{n}{3}) + \frac{1}{n} \\ &= 3 \cdot \left(3 \cdot T(\frac{n}{3^2}) + \frac{3}{n}\right) + \frac{1}{n} \\ &= 3^2 \cdot T(\frac{n}{3^2}) + \frac{3^2}{n} + \frac{3^0}{n} \\ &= 3^2 \cdot \left(3 \cdot T(\frac{n}{3^3}) + \frac{3^2}{n}\right) + \frac{3^2}{n} + \frac{3^0}{n} \\ &= 3^3 \cdot T(\frac{n}{3^3}) + \frac{3^4}{n} + \frac{3^2}{n} + \frac{3^0}{n} \\ &= \ldots \end{aligned}$$

Ein weitere Beispiel

$$
\begin{aligned}
T(n) &= 3^3 \cdot T(\frac{n}{3^3}) + \frac{3^4}{n} + \frac{3^2}{n} + \frac{3^0}{n} \\
&= \dots \\
&= 3^k \cdot T(\frac{n}{3^k}) + \frac{1}{n} \sum_{i=0}^{k-1} 3^{2i} \ (*) \\
&= c \cdot n + \frac{1}{n} \frac{3^{2k} - 1}{3^2 - 1} \ (\text{mit } k = \log_3(n) \text{ bzw. } n = 3^k) \\
&= c \cdot n + \frac{n^2 - 1}{8n} = c \cdot n + \frac{1}{8} \left(n - \frac{1}{n} \right)
\end{aligned}
$$

(*) Diese Zeile ist wieder mit vollständiger Induktion zu zeigen!

Ein weitere Beispiel

Aus

$$T(n) = c \cdot n + \frac{1}{8}\left(n - \frac{1}{n}\right)$$

folgt nun

- $T(n) < (c + \frac{1}{8}) \cdot n$ und damit $T(n) \in O(n)$ und
- $T(n) > c \cdot n$ und damit $T(n) \in \Omega(n)$.

Insgesamt haben wir also

$$T(n) \in \Theta(n)$$

gezeigt!

Mastertheorem

Definition (Mastertheorem)

Das Mastertheorem gilt für Rekurrenzen der Form:

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & \text{falls } n > 1 \end{cases}$$

Hierbei sind $a \geq 1, b > 1$ und c Konstanten, $\frac{n}{b}$ wird als $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ verstanden und $T(n)$ über $\mathbb{N}$ definiert. Es gilt:

1. $T(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.
2. $T(n) \in \Theta(n^{\log_b(a)} \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$.
3. $T(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n.

Mastertheorem - Beispiel 1

$$T(n) = \begin{cases} c' \\ 3T(n/3) + 1/n \end{cases} \qquad T(n) = \begin{cases} c \\ a \cdot T(\frac{n}{b}) + f(n) \end{cases}$$

Wegen $\log_b(a) = \log_3(3) = 1$ und $f(n) = \frac{1}{n}$ ist $f(n) \in O(n^{1-\epsilon})$ bei Wahl von z.B. $\epsilon = 2 > 0$.

Es liegt also Fall 1 vor und dies ergibt, wie vorher durch Abwickeln, ebenfalls $T(n) \in \Theta(n)$.

Mastertheorem - Anmerkung

Anmerkung

Das Mastertheorem ist die **dritte Methode**, die wir zur Lösung von Rekurrenzgleichungen nutzen werden.
Das Mastertheorem ist ein mächtiges Hilfsmittel. Es ist aber nicht immer einsetzbar! Z.B. bei $T(n) = 2 \cdot T(n/2) + n \cdot \log n$ nicht.

Bemerkung

Wir hatten hier nur die zweite und dritte Methode. Die erste Methode war einsetzen, raten und beweisen. Siehe den Foliensatz dazu.

Suchen und Sortieren

Zusammenfassung:

Suchen und Sortieren

Sortieren

Definition (Das Sortierproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen.
Gesucht: Eine Permutation $< a'_1, a'_2, \ldots, a'_n >$ der Eingabesequenz mit $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Anmerkung

Ob die Reihenfolge zweier Elemente a_i, a_j $(i < j)$ mit $a_i = a_j$ beibehalten werden soll oder nicht, ist nicht gesagt. Bleibt sie erhalten, d.h. ist $a'_p = a_i$, $a'_q = a_j$ und $p < q$, so nennt man das Verfahren *stabil*.
Ferner heißt ein Sortierverfahren *in-place* (oder *in situ*), wenn der zusätzlich benötigte Speicherbedarf unabhängig von der gegebenen Sequenz ist.

Datentypen und Datenstrukturen

Zusammenfassung:

Datentypen und Datenstrukturen

(kurzer Einschub)

Datentypen

Elementare und strukturierte Datentypen

- Elementare Datentypen sind z.B. integer, boolean, char, ...
- Strukturierte Datentypen sind z.B. Arrays und Records, ...

Datenstrukturen: Stack

Stack

- Zugriff: LIFO
- Operationen: head, push, pop, (empty)
- Implementierung:
  - Sequentiell (Array)
    - Problematisch bei Größenänderungen
  - Verkettete Liste (s.u.)
- Anwendung: Erkennen wohlgeformter Klammerausdrücke
  - Ermitteln der zusammengehoerigen Klammerpaare in $O(n)$ (Idee: Index der öffnenden Klammer auf Stack pushen)
  - Noch schneller geht's ohne Stack indem man einen Zähler bei öffnenden Klammern um eins erhöht und bei schließenden Klammern um eins verringert.

Datenstrukturen: Queue

Warteschlange

- Zugriff: FIFO
- Operationen: head, enqueue, dequeue, (empty)
- Implementierung:
  - Sequentiell (Array, ggf. zyklisch!)
    - Problematisch bei Größenänderungen
  - Verkettete Liste (s.u.)
- Anwendung: Prioritätswarteschlangen

Datenstrukturen: Listen

Listen

- Ähnelt einem Array
- Zugriff: Random (Dauer abhängig von Implementation)
- Operationen: insert(x,p), remove(p), search(x), lookup(p), empty, concat, lenght, append, head, tail, ...
- Implementierung:
  - Sequentiell (Array)
    - \+ schneller Zugriff
    - – langsames Einfügen
  - Verkettete Speicherung
    - \+ schnelles Einfügen/Löschen
    - – langsamer Zugriff, höherer Speicherbedarf
- *Verkettung* kann *einfach* oder *doppelt* sein; letzteres erlaubt auch Durchlaufen von hinten nach vorne.
- *Wächter* vereinfachen die Implementierung.

Datenstrukturen: Bäume

Bäume

- Begriffe: Knoten, Vorgänger, Nachfolger, Wurzel, innerer Knoten, Blatt Höhe, Tiefe, Ebene ($\approx$ Tiefe)
- Speziell: Binärbaum
  - Anzahl Knoten auf Ebene i: 2^i
  - Anzahl Blätter: 2^h (h ist die Höhe)
  - Anzahl Knoten: $|T| = \sum_{i=0}^{h} = 2^i = 2^{h+1} - 1$
  - Zur Speicherung von $|T|$ Knoten braucht man einen Binärbaum der Höhe $\log_2(|T| + 1) - 1 \in \Theta(\log(|T|))$, soll heissen ca. $\log_2(|T|)$ viele.

Suchen und Sortieren

Zusammenfassung:

Suchen und Sortieren

(Fortsetzung)

Verschiedene Sortierverfahren

- InsertionSort
- Min/MaxSort (= SelectionSort)
- BubbleSort
- MergeSort
- QuickSort
- HeapSort
- CountingSort
- (RadixSort, BucketSort)

Anmerkung

Vergleiche mit der anfänglichen Bemerkung: Eingabe? Ausgabe? Idee? Laufzeit? Ungefährer Ablauf?

Buzzwords

Nett zu wissen:

- Vergleichendes Sortieren?
- Nicht-vergleichendes Sortieren?
- Untere Schranke für vergleichendes Sortieren?

Heaps

Ein (binärer) *Heap* ist eine Datenstruktur, die als ein (fast) vollständiger binärer Baum angesehen werden kann, wobei der Baum noch die spezielle Heap-Eigenschaft erfüllt.

- Der Baum ist auf allen Ebenen vollständig gefüllt außer möglicherweise auf der letzten.
- Der Heap wird meist durch ein Array repräsentiert.
  - *länge*$[A]$ ist die Anzahl der Elemente *des Feldes*
  - *heap-größe*$[A]$ ist die Anzahl der Elemente *im Heap* (die in A gespeichert werden).
- *Heap-Eigenschaft*: Sei n ein Knoten des Baumes und p sein Vater, dann gilt
  - Max-Heap: $A[Vater(i)] \geq A[i]$
  - Min-Heap: $A[Vater(i)] \leq A[i]$

$\Rightarrow$ Gut für Sortieren und Warteschlangen.

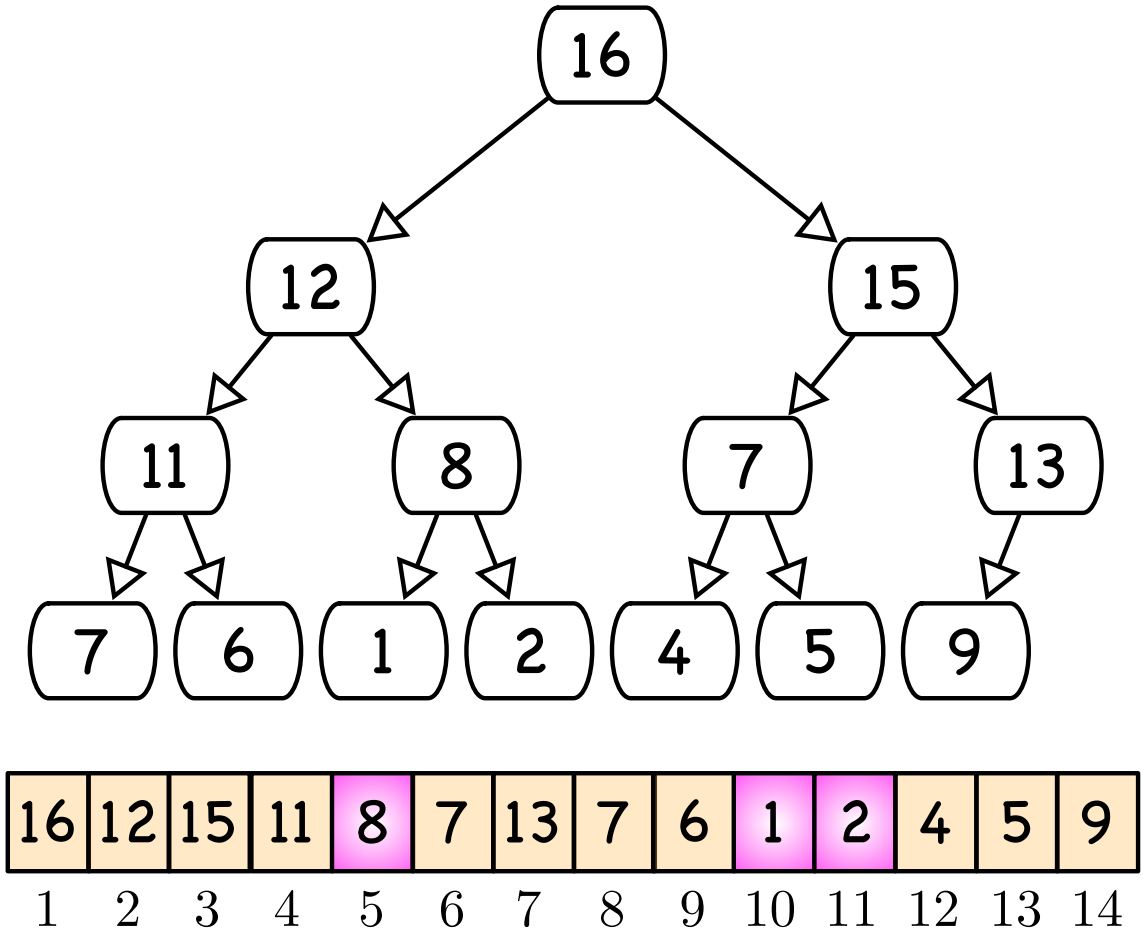
Heaps und Arrays

Hat man ein Array, so werden die Elemente von links nach rechts gelesen und der binäre Baum von oben nach unten und von links nach rechts Ebene für Ebene aufgebaut.

Hat man einen Baum, so wird dieser von oben nach unten und von links nach rechts gelesen und dabei das Array von links nach rechts gefüllt.

$$\begin{aligned} \mathit{Vater}(i) &= \lfloor i/2 \rfloor \\ \mathit{Left}(i) &= 2i \\ \mathit{Right}(i) &= 2i+1 \end{aligned}$$

Heap - Beispiel



Heaps - Zusammenfassung

- Heaps - Ziel: Sortieren
  - Definition, Array- und Baumdarstellung
  - Vater(i), Left(i), Right(i)
  - MaxHeapify zur Aufrechterhaltung der Heap-Eigenschaft
  - BuildMaxHeap zur Erstellung eines MaxHeaps (benutzt MaxHeapify)
  - HeapSort sortiert ein Array (benutzt MaxHeapify und BuildMaxHeap)
- Heaps - Warteschlangen
  - HeapMaximum und HeapExtractMax (letztere nutzt wieder MaxHeapify)
  - HeapIncreaseKey (im Prinzip die 'Umkehrung' von MaxHeapify)
  - HeapInsert (nutzt HeapIncreaseKey)

Analysen

Satz

1. *MaxHeapify ist korrekt und hat eine Laufzeit von* $\Theta(\log n)$*, wobei* n *die Anzahl der Knoten des Baumes ist.*
2. *BuildMaxHeap ist korrekt und die Laufzeit ist durch* $O(n \cdot \log n)$ *beschränkt. Eine genauere Analyse zeigt, dass die Laufzeit sogar durch* $O(n)$ *beschränkt ist. Wir können einen Heap also in Linearzeit herstellen.*
3. *HeapSort sortiert ein Array von* n *Zahlen in* $\Theta(n \cdot \log n)$ *Schritten. HeapSort sortiert in-place.*
4. *Die Operationen HeapMaximum, HeapExtractMax, HeapIncreaseKey und HeapInsert sind alle korrekt und arbeiten in* $O(\log n)$ *Zeit. HeapMaximum sogar in* $\Theta(1)$*.*

Das Suchproblem

Definition (Das Suchproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen und ein Wert k.
Gesucht: Ein Index i mit $k = A[i]$, falls k in A vorkommt oder der Wert `nil`.

Definition (Das Suchproblem (allgemein))

Eingabe: Eine Menge M und ein Wert k.
Gesucht: Ein Zeiger x auf ein Element in M mit schlüssel$[x] = k$ oder `nil`, falls kein solches Element existiert.

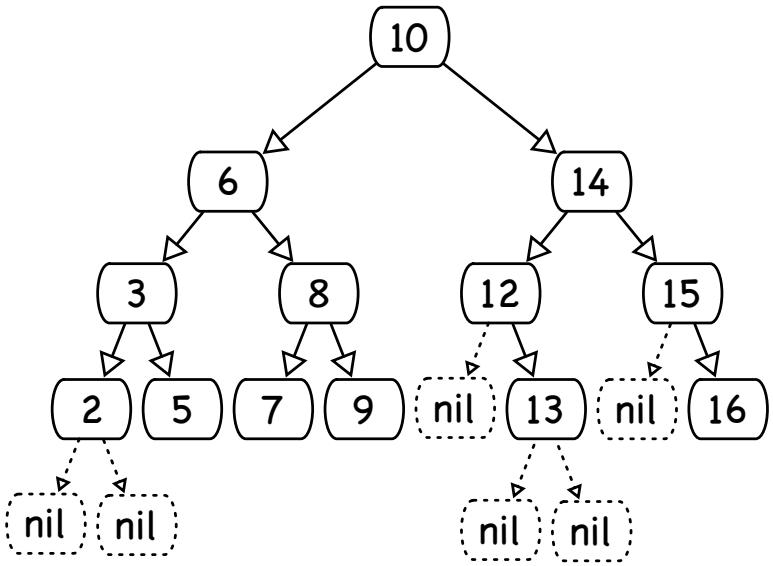
Binäre Suchbäume - Die Idee

Ein *binärer Suchbaum* ist eine Datenstruktur, die viele Operationen auf dynamischen Mengen ermöglicht, insb. die Operationen Search, Minimum, Maximum, Predecessor, Successor, Insert und Delete. Ein binärer Suchbaum ist ein binärer Baum (jeder Knoten hat bis zu zwei Kindern), wobei der Baum noch spezielle Suchbaum-Eigenschaft erfüllt.

- Der Baum wird (anders als der Heap!) meist durch eine verkettete Datenstruktur (ähnlich der doppelt-verketteten Liste) repräsentiert. Jeder Knoten ist ein Objekt mit den Attributen
  - `schlüssel` (und den Satellitendaten),
  - `links`, `rechts`, `p`, die auf den jeweiligen linken bzw. rechten Nachfolger, bzw. den Vaterknoten zeigen (fehlen diese, sind die entsprechenden Attribute auf `nil` gesetzt).

Binäre Suchbäume - Die Idee

- *Binäre Suchbaum-Eigenschaft*: Sei p ein Knoten im binären Suchbaum. Sei l ein Knoten im linken Teilbaum von n, r im rechten, dann gilt stets: $\texttt{schlüssel}[l] \leq \texttt{schlüssel}[p]$ und $\texttt{schlüssel}[p] \leq \texttt{schlüssel}[r]$.



Dynamische Mengen

Wichtige Operationen (dynamischer Mengen):

- Search(S, k). Gibt es einen Zeiger x auf ein Element in S mit schlüssel$[x] = k$?
- Insert(S, x).
- Delete(S, x).
- Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x).

Viele hiervon haben wir schon bei z.B. der verketteten Liste gesehen, aber wie schnell waren die?

Operationen

Zusammenfassung der Operationen

Die Operationen InorderTreeWalk, TreeSearch, Minimum, Maximum, Successor, Insert und Delete sind alle *korrekt*, was man stets recht einfach über die Eigenschaften binärer Suchbäume zeigen kann (ggf. mit Induktion über die Anzahl der Knoten oder die Tiefe des Baumes).
Bei einem binären Suchbaum mit n Knoten der Höhe h, ist die Laufzeit von InorderTreeWalk in $O(n)$, die aller anderen Operationen in $O(h)$.

Anmerkung

Diese Laufzeiten sind zwar gut, da bei einem (ungefähr) ausgeglichenen Baum $h \approx \log n$ ist. Das Problem ist aber, dass binäre Suchbäume nicht ausgeglichen sein müssen! Im worst-case sind obige Operationen alle in $\Theta(n)$!

Rot-Schwarz-Bäume - Die Idee

Rot-Schwarz-Bäume sind binäre Suchbäume, bei denen jeder Knoten eine 'Farbe' hat (rot oder schwarz). **Durch Einschränkungen (Rot-Schwarz-Eigenschaften) wird sichergestellt, dass der Baum annähernd balanciert ist**, so dass die Operationen dann auch im worst-case in $O(\log n)$ laufen.

Das Problem ist es dann die Operationen (insb. Einfügen und Löschen) so zu implementieren, dass die Rot-Schwarz-Eigenschaften erhalten bleiben!

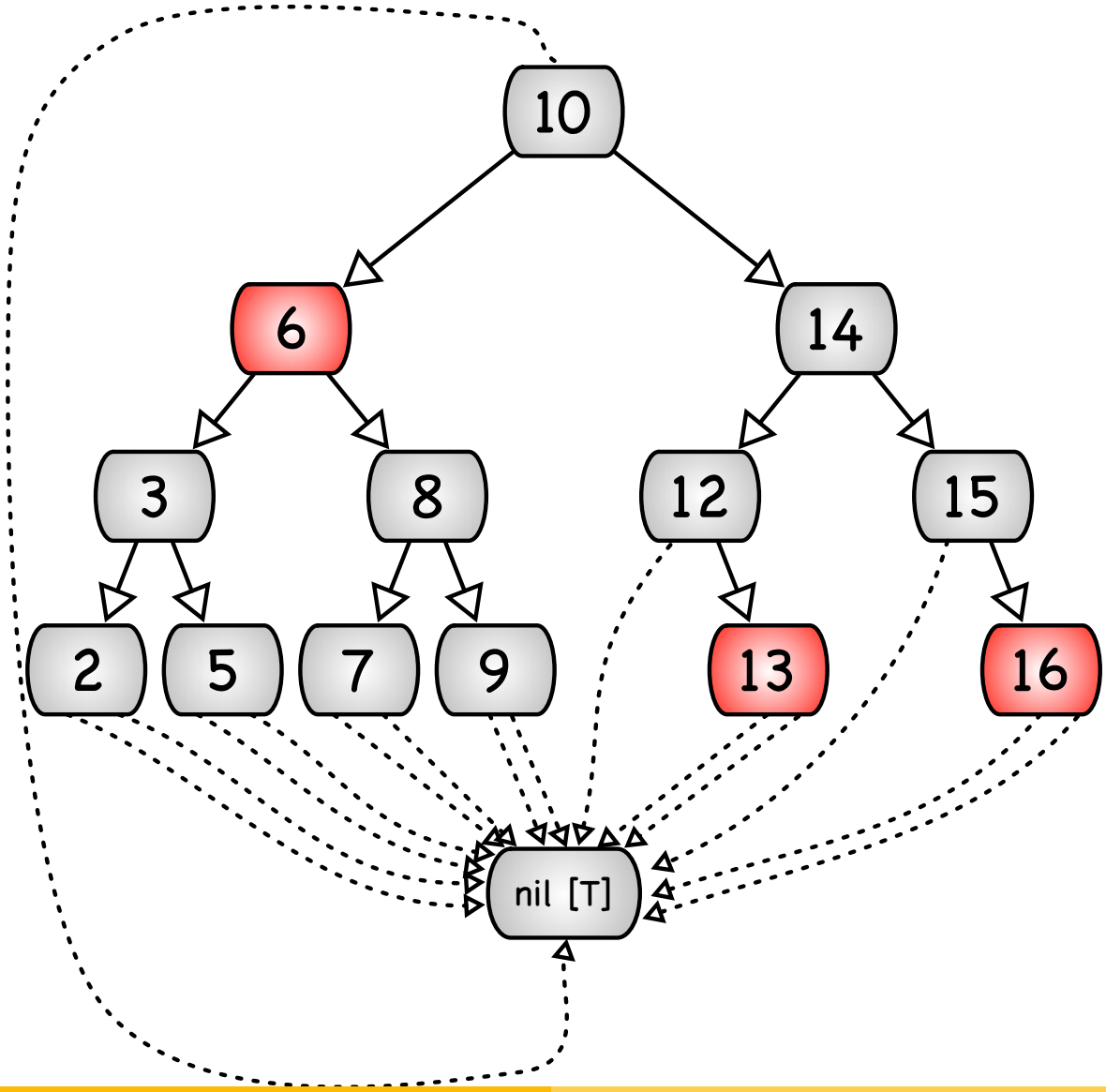
Rot-Schwarz-Eigenschaften (Wiederholung)

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ist schwarz.
4. *Wenn ein Knoten rot ist, dann sind seine beiden Kinder schwarz.*
5. *Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten*

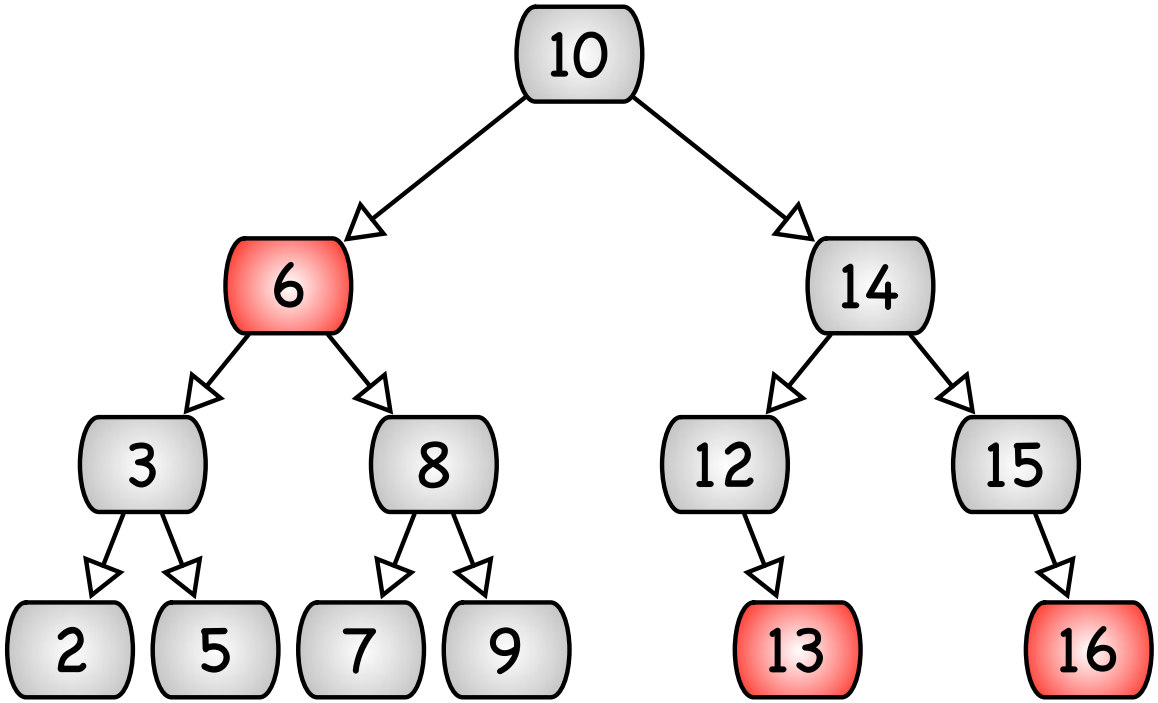
Anmerkung

Die Blätter sind *alle* `nil`-Knoten (der Wächter `nil`$[T]$) und schwarz. Die Wurzel hat als Vorgänger auch `nil`$[T]$ (Siehe die beiden vorherigen Bilder.)

Ein Rot-Schwarz-Baum (gefärbt)



Ein Rot-Schwarz-Baum (in schön)



R-S-Bäume - Zusammenfassung

Zusammenfassung

- Rot-Schwarz-Bäume sind binäre Suchbäume, die die zusätzlichen Rot-Schwarz-Eigenschaften erfüllen.
- Ein Baum der die Rot-Schwarz-Eigenschaften erfüllt ist in etwa ausgeglichen ($h \leq 2 \cdot \log(n+1)$).
- Der Erhalt der Rot-Schwarz-Eigenschaften ist insb. bei der Einfüge- und der Löschoperation zu beachten. Sie können in $O(\log n)$ implementiert werden.
- Damit erlauben Rot-Schwarz-Bäume die wichtigen Operationen für dynamische Mengen alle in $O(\log n)$ Zeit!
- (Allerdings sind sie dafür aufwändiger als normale binäre Suchbäume.)

Graphen

Zusammenfassung:

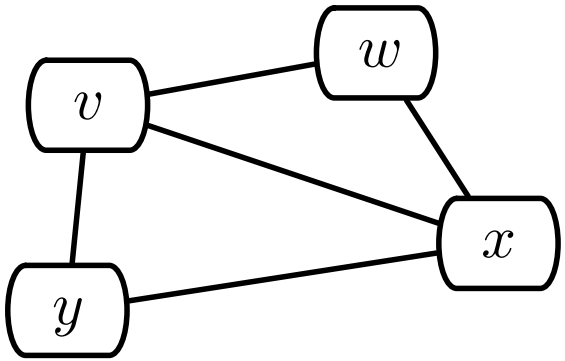
Graphen

Graphen - Definitionen

Definition

Graph, Knoten, Kante, ungerichteter Graph, gerichteter Graph, dünn/dicht besetzt, vollständig, unabhängig, Grad, Nachbarn, Minimalgrad, Maximalgrad, Weg, Kreis, Länge (eines Weges/Kreises), Abstand, Teilgraph, induzierter (Teil-)graph, K^n, Baum, ...

Darstellung von Graphen - Adjazenzmatrix

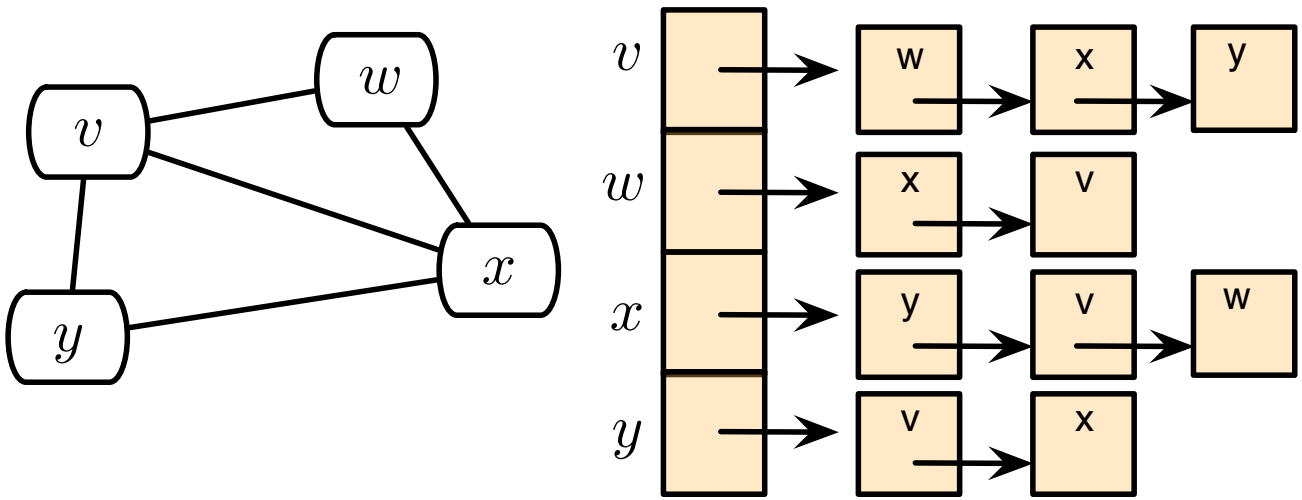


	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

Bei einer Adjazenzmatrix hat man eine $n \times n$-Matrix, bei der an der Stelle (i, j) genau dann eine 1 steht, wenn v_i und v_j verbunden sind.

Der Speicherplatzbedarf ist in $\Theta(V^2)$ (unabhängig von der Kantenzahl).

Darstellung von Graphen - Adjazenzlisten



Bei der Adjazenzlistendarstellung haben wir ein Array von $|V|$ Listen, für jeden Knoten eine. Die Adjazenzliste $Adj[v]$ zu einem Knoten v enthält alle Knoten, die mit v adjazent sind.

Bei einem gerichteten Graphen ist die Summe aller Adjazenzlisten $|E|$, bei einem ungerichteten Graphen $|2E|$. Der Speicherplatzbedarf ist folglich $\Theta(V + E)$.

Breitensuche - Initialisierung

Algorithmus 1 BFS(G, s) - Teil 1, Initphase

1: **for** each $u \in V(G) \setminus \{s\}$ **do**
2: farbe$[u]$ = weiss, $d[u] = \infty, \pi[u] = \texttt{nil}$
3: **end for**
4: farbe$[s]$ = grau
5: $d[s] = 0, \pi[s] = \texttt{nil}$
6: $Q = \emptyset$, enqueue(Q, s)

Anmerkung

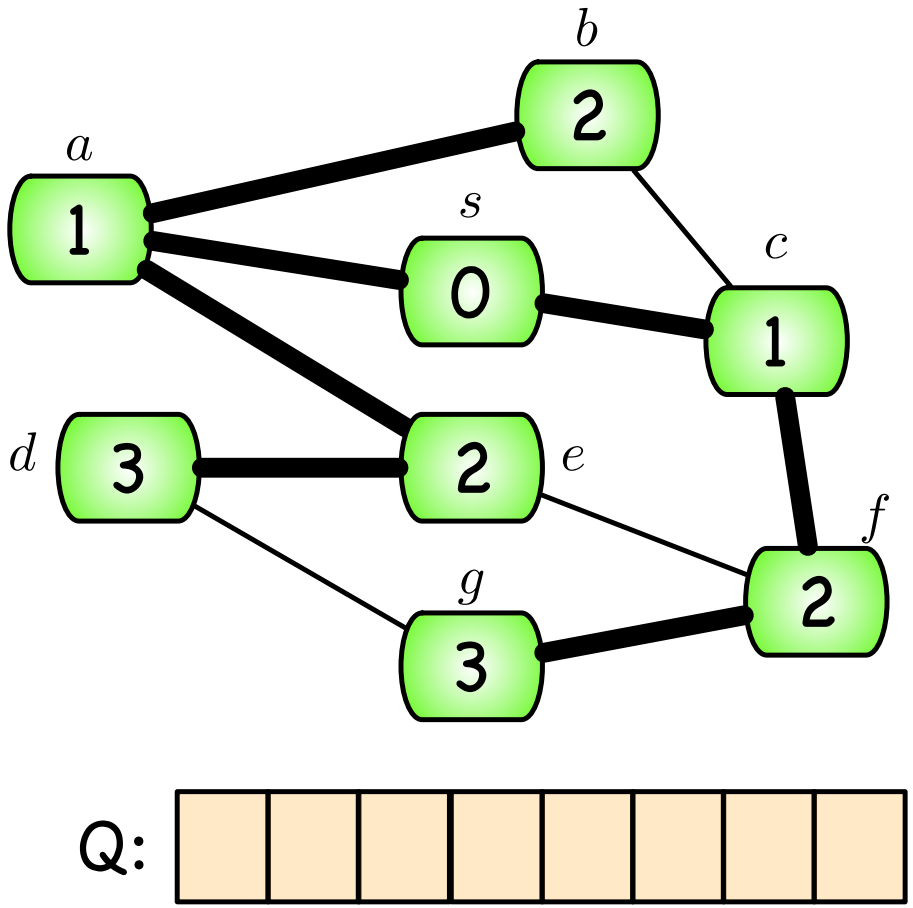
Farben: weiss heißt 'noch nicht besucht', grau 'besucht, aber noch unbesuchte Nachbarn', schwarz 'besucht, alle Nachbarn entdeckt'. $d[u]$ ist die Anzahl der Schritt von s zu u, $\pi[u]$ der Vorgänger von u auf diesem Pfad.

Breitensuche - Hauptschleife

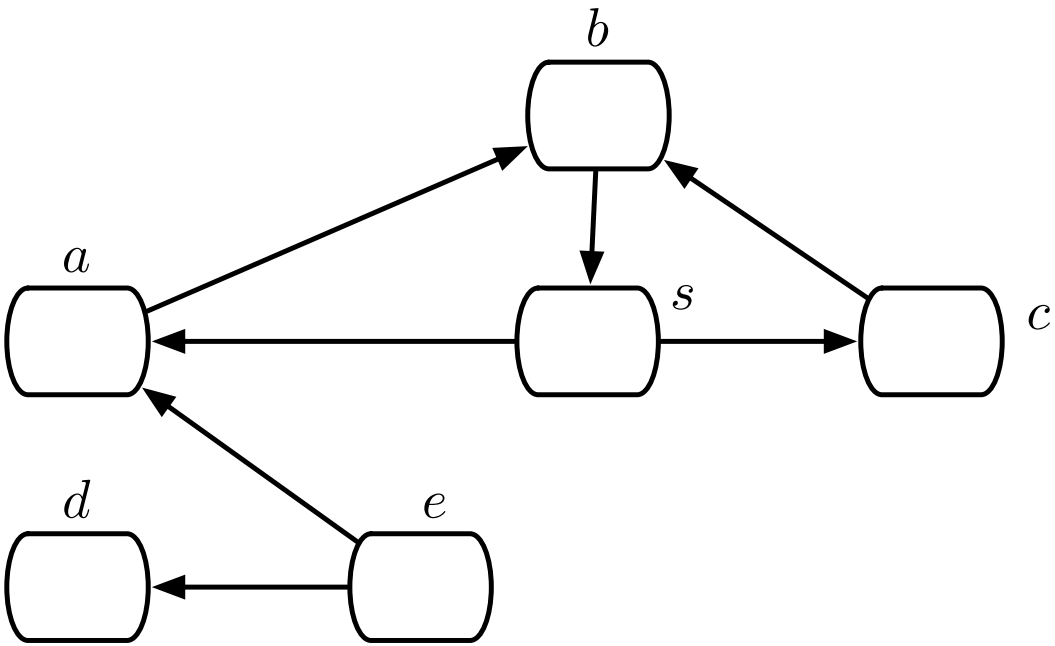
Algorithmus 2 BFS(G, s) - Teil 2, Hauptteil

```
 1: while Q ≠ ∅ do
 2:     u = dequeue(Q)
 3:     for each v ∈ Adj[u] do
 4:         if farbe[v] == weiss then
 5:             farbe[v] = grau
 6:             d[v] = d[u] + 1, π[v] = u
 7:             enqueue(Q, v)
 8:         end if
 9:     end for
10:     farbe[u] = schwarz
11: end while
```

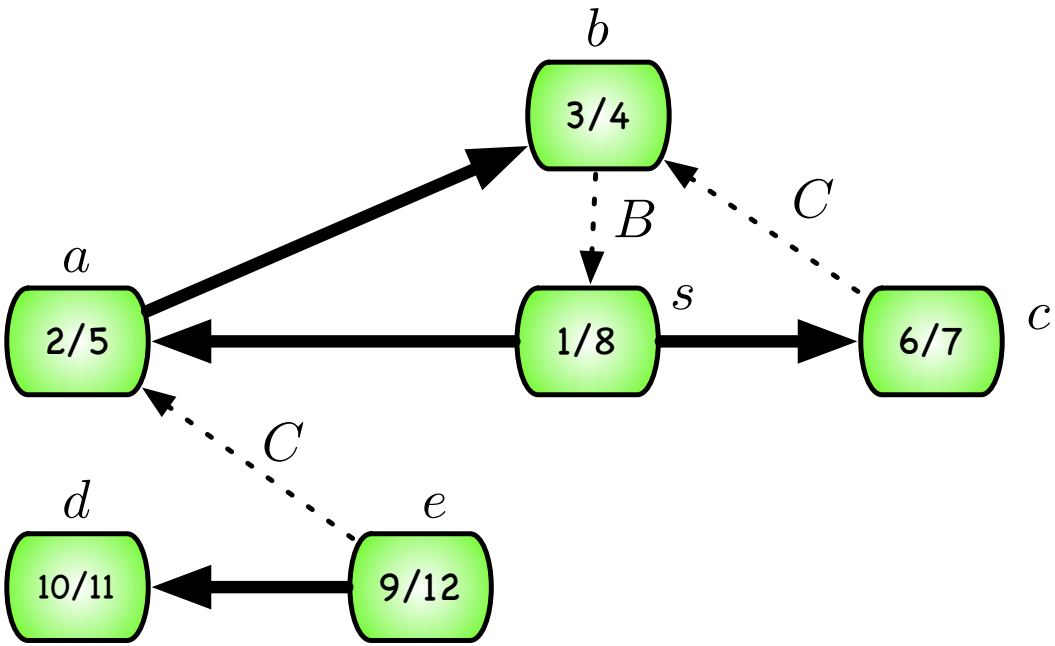
Breitensuche - Beispiel



Tiefensuche - Beispiel



Tiefensuche - Beispiel



Problemstellung

Definition (Topologisches Sortieren)

Eingabe: Gegeben ein gerichteter azyklischer Graph $G = (V, E)$.
Gesucht: Eine *topologischer Sortierung*, d.h. eine lineare Anordnung der Knoten mit der Eigenschaft, dass u in der Anordnung vor v liegt, falls es in G eine Kante (u, v) gibt.

Anmerkung

Dieses Problem tritt häufig bei der Modellierung von Ereignissen auf, die mit einer Priorität (bzw. Abhängigkeiten) versehen sind (und entsprechend bearbeitet werden sollen).

TopoSort - Algorithmus und Idee

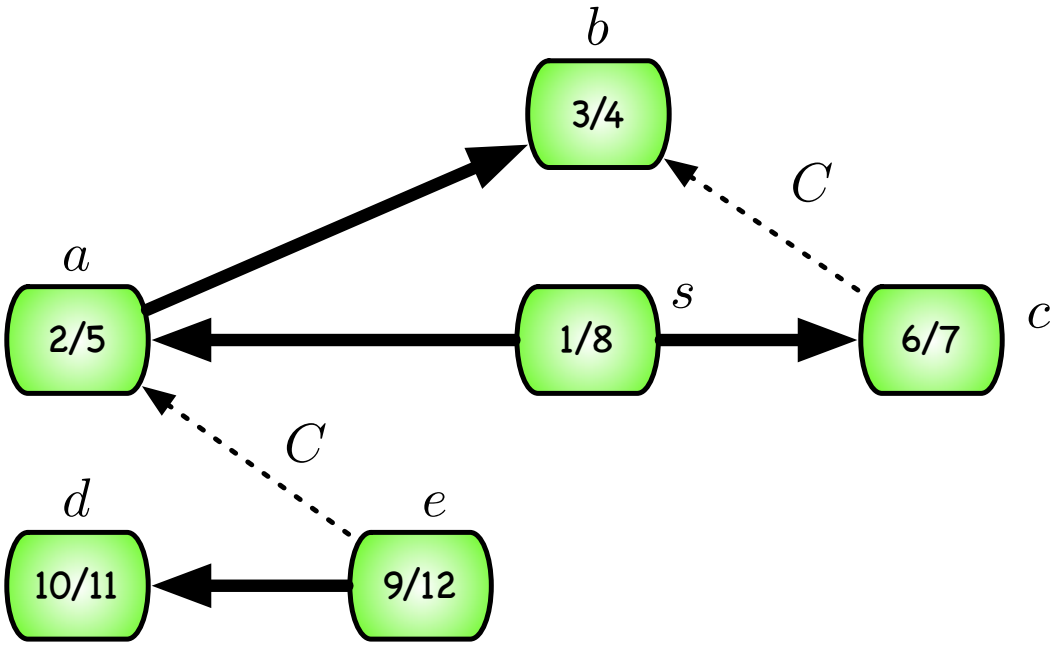
Algorithmus 3 TopoSort(G)

1: Berechne die Endzeiten $f[v]$ für alle $v \in V$ durch Aufruf von DFS(G).
2: Füge jeden abgearbeiteten Knoten an den Kopf einer verketteten Liste ein.
3: Liefere die verkettete Liste zurück.

Anmerkung

Die Kernidee ist es die Knoten in umgekehrter Reihenfolge ihrer Abarbeitungszeiten auszugeben (d.h. der Knoten, der zuletzt abgearbeitet wurde, wird zuerst ausgegeben usw.).

TopoSort - Beispiel



e, d, s, c, a, b

Problemstellung

Definition (Bestimmung starker Zusammenhangskomponenten)

Eingabe: Gegeben ein gerichteter Graph $G = (V, E)$.
Gesucht: Die starken Zusammenhangskomponenten (SCCs) des Graphen, d.h. maximale Menge $C \subseteq V$ derart, dass für jedes Paar $u, v \in C$ sowohl $u \Rightarrow v$ als auch $v \Rightarrow u$ gilt (es gibt einen Pfad von u nach v und andersherum).

Anmerkung

Dieses Problem tritt oft bei gerichteten Graphen auf. Zunächst wird der Graph in seine starken Zusammenhangskomponenten zerlegt und dann werden diese separat betrachtet. (Oft werden die Lösungen anschließend noch entsprechend der Verbindungen zwischen den einzelnen Komponenten zusammengefügt.)

SCC - Algorithmus und Idee

Algorithmus 4 SCC(G)

1: Aufruf von DFS(G) zur Berechnung der Endzeiten $f[v]$.
2: Berechne G^T (transponierter Graph).
3: Aufruf von DFS(G^T), betrachte in der Hauptschleife von DFS die Knoten jedoch in der Reihenfolge fallender $f[v]$ (in Zeile 1 berechnet).
4: Die Knoten jedes in Zeile 3 berechneten Baumes sind eine (separate) strenge Zusammenhangskomponente.

Anmerkung

Der zu einem Graphen $G = (V, E)$ transponierte Graph ist definiert durch $G^T = (V, E^T)$ mit $E^T = \{(u, v) \mid (v, u) \in E\}$. Ist G in Adjazenzlisten-Darstellung gegeben, kann G^T in Zeit $O(V + E)$ berechnet werden.

Laufzeitanalyse

Satz

Ein gerichteter Graph G ist genau dann azyklisch, wenn eine Tiefensuche auf G keine Rückwärtskanten liefert.

Satz

`TopoSort`(G) *ist korrekt und arbeitet in* $\Theta(V + E)$.

Satz

`SCC`(G) *ist korrekt und arbeitet in* $\Theta(V + E)$.

Beweis.

Der Beweis jedes der obigen Sätze nutzt Eigenschaften der Tiefensuche und insb. die dort ermittelten Endzeiten $f[v]$. □

Problemstellung

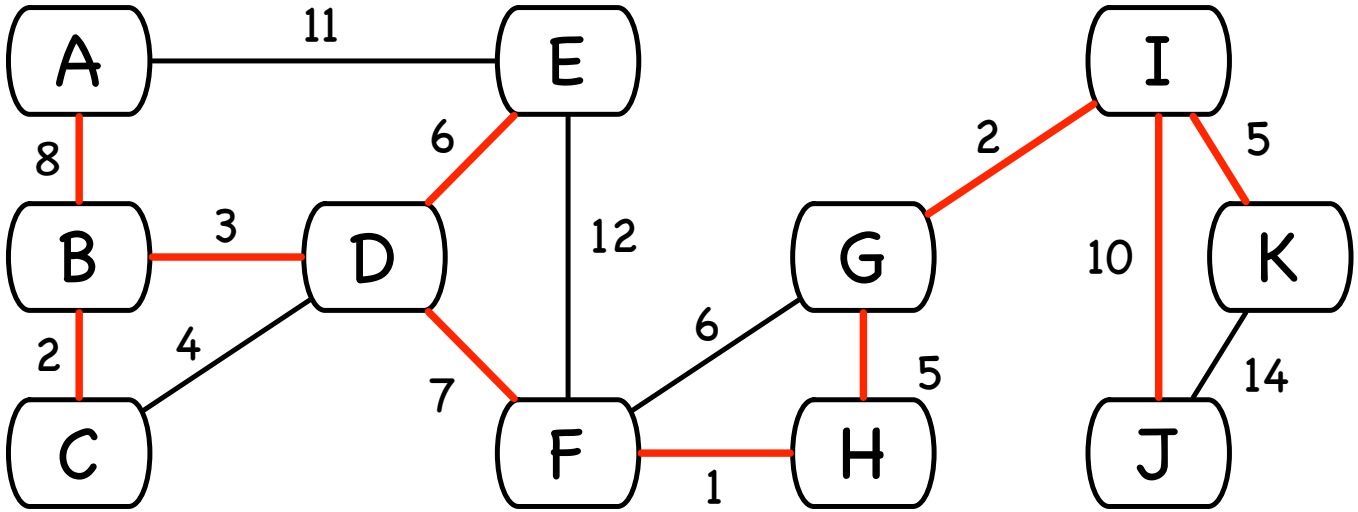
Definition (Bestimmung eines minimalen Spannbaums)

Eingabe: Gegeben ein ungerichteter Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to \mathbb{R}^+$.
Gesucht: Ein Spannbaum T derart, dass

$$w(T) = \sum_{(u,v) \in E(T)} w(u, v)$$

minimal ist. Ein Spannbaum ist dabei ein Graph, der zusammenhängend und azyklisch ist (also ein Baum) und alle Knoten von G enthält.

Ein Beispiel



Anmerkung

Alle Spannbäume haben $|V| - 1$ Kanten. Es ist oben also wichtig, dass das summierte Gewicht dieser Kanten minimiert werden soll. Breitensuche und Tiefensuche liefern also bereits minimale Spannbäume bei ungewichteten Graphen.

Kruskal und Prim

Sei A die bisher gewählte Menge von Kanten für den MST. Die Algorithmen von Kruskal und Prim arbeiten wie folgt:

- Kruskal: Nimm unter allen Kanten (aus $E - A$) jene mit dem kleinsten Gewicht, deren Hinzufügen nicht zu einem Kreis führt.
- Prim: Wähle aus den Kanten eine minimale aus, die A mit einem noch isolierten Knoten von G_A verbindet.

Korrektheit relativ kompliziert zu zeigen.

Kruskal und Prim - Zusammenfassung

Beide vorgestellten Algorithmen (Kruskal und Prim) verwenden eine bestimmte Regel, um die sichere Kante zu bestimmen.

- Kruskal: Die Menge A ist ein Wald. Die hinzugefügte, sichere Kante ist immer eine Kante mit minimalen Geweicht, die zwei verschiedene Komponenten (von G_A) verbindet.
- Prim: Die Menge A ist ein einzelner Baum. Die hinzugefügte, sichere Kante ist immer eine Kante mit minimalen Gewicht, die den Baum mit einem Knoten verbindet, der noch nicht zum Baum gehörte.
- Beide Algorithmen sind *nicht eindeutig*.
- Die Algorithmen können verschiedene Ergebnisse liefern.

Mit guten Datenstrukturen ist Kruskal in $O(E \cdot \log V)$ möglich, Prim ist in $O(E \cdot \log V)$ bei Nutzung von Min-Heaps und sogar in $O(E + V \cdot \log V)$ bei Nutzung von Fibonacci-Heaps.

Problemstellung

Definition (Problem der kürzesten Pfade)

Eingabe: Gegeben ein gerichteter Graph $G = (V, E)$, eine Gewichtsfunktion $w : E \to \mathbb{R}^+$ und ein Knoten s.
Gesucht: Für jeden Knoten $v \in V$ ein Pfad P_v, der in s beginnt, in v endet und der unter allen Pfaden mit dieser Eigenschaft das geringste Gewicht hat.

Anmerkung

Das Gewicht eine Pfades $p = [v_0, v_1, \ldots, v_k]$ ist dabei definiert als $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

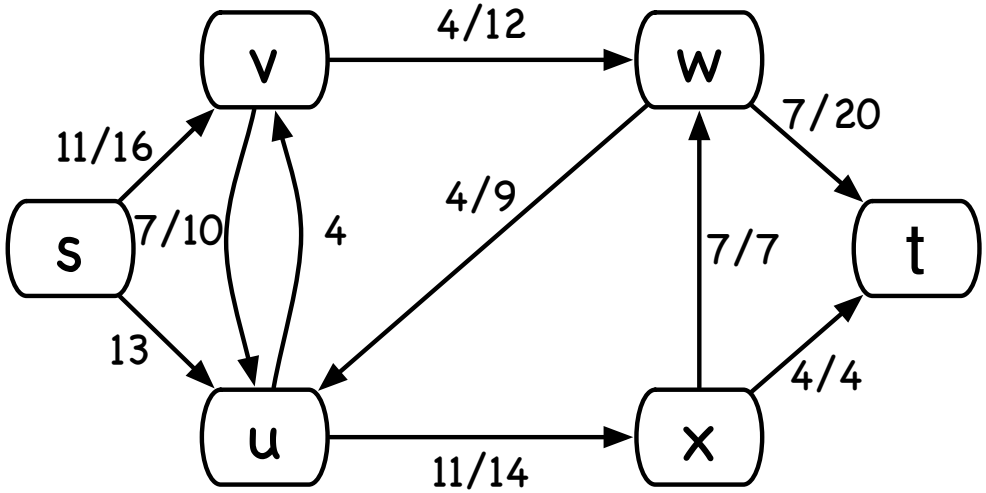
Kürzeste Pfade - Zusammenfassung

- Dijkstra. Für gewichtete, gerichtete Graphen, *ohne* negative Gewichte. Laufzeit: $O(V^2)$ und sogar $O(V \cdot \log V + E)$.
- Bellman-Ford(-Moore). Für gewichtete, gerichtete Graphen. Negative Gewichte sind erlaubt. Laufzeit: $O(V \cdot E)$.
- Floyd-Warshall. Für gewichtete, gerichtete Graphen. Negative Gewichte sind erlaubt. Laufzeit: $O(V^3)$. Ermittelt anders als die obigen beiden kürzeste Pfade *für alle Knotenpaare*!

Anmerkung

Alle obigen Verfahren gehen auch mit ungerichteten Graphen (da eine ungerichtete Kante durch zwei gerichtete Kanten ersetzt werden kann), sowie für ungewichtete Graphen (allen Kanten Gewicht 1 geben).

Begriffe und Definitionen



Definition (Flussnetzwerk)

Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$, in dem jeder Kante $(u, v) \in E$ zusätzlich eine nichtnegative *Kapazität* $c(u, v) \geq 0$ zugewiesen wird. (Ist $(u, v) \notin E$, so nehmen wir $c(u, v) = 0$ an.)
Es gibt die besonderen Knoten $s, t \in V$, die *Quelle* (s) und die *Senke* (t). Alle Knoten liegen auf Pfaden von der Quelle zur Senke.

Begriffe und Definitionen

Definition (Fluss)

Sei $G = (V, E)$ mit Kapazitätsfunktion c ein Flussnetzwerk, s und t wie oben. Ein *Fluss* in G ist eine Funktion $f : V \times V \to \mathbb{R}$, für die gilt:

1. Kapazitätsbeschränkung: $f(u, v) \leq c(u, v) \quad \forall u, v \in V$
2. Asymmetrie: $f(u, v) = -f(v, u) \quad \forall u, v \in V$
3. Flusserhaltung:

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

Das Problem des maximalen Flusses

Definition (Problem des maximalen Flusses)

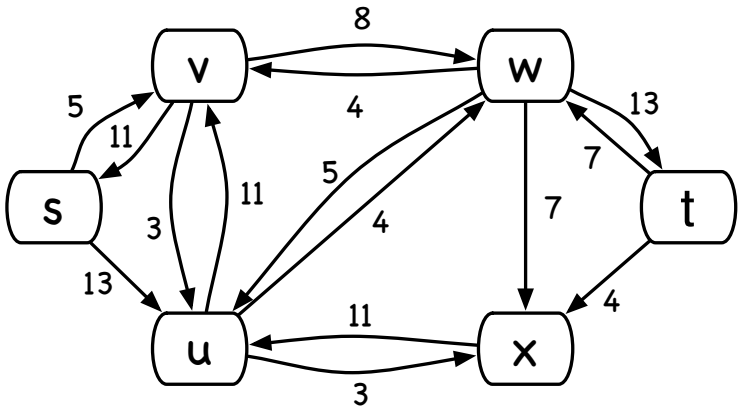
Eingabe: Ein Flussnetzwerk G mit Kapazitätsfunktion c und Quelle s und Senke t.
Gesucht: Ein Fluss mit maximalen Wert. Dabei ist der *Wert eines Flusses* definiert durch

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

Anmerkung

Dieses Problem tritt bei vielen Anwendungen (Logistik, Material-/Informationsflüsse, ...) direkt auf, oft lassen sich ganz anders erscheinende Probleme aber auch als Flussprobleme formulieren und dann so lösen.

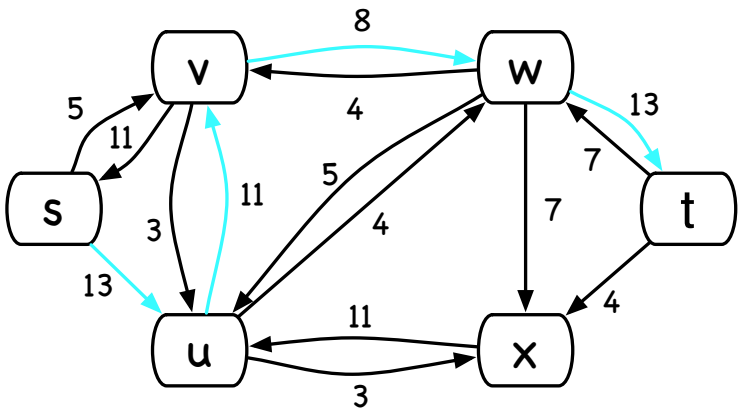
Restnetzwerke: Definition



Definition

Sei $G = (V, E)$ mit c ein Flussnetzwerk und f ein Fluss in G. Wir definieren zu je zwei Knoten u, v die *Restkapazität* durch $c_f(u, v) = c(u, v) - f(u, v)$. Das Restnetzwerk ist dann gegeben durch $G_f = (V, E_f)$ und c_f, wobei $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$ die Menge der *Restkanten* ist. Das Restnetzwerk besteht also aus jenen Kanten, die noch mehr Fluss aufnehmen können.

Erweiterungspfade: Definition



Definition

Sei $G = (V, E)$ mit c ein Flussnetzwerk, f ein Fluss in G und G_f das Restnetzwerk. Ein *Erweiterungspfad* ist ein einfacher Pfad von s nach t in G_f. Mit

$$c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ ist eine Kante von } p\}$$

wird die *Restkapazität* eines Erweiterungspfades p bezeichnet.

Restnetzwerke und Erweiterungspfade: Zusammenfassung

- Intuitive Definition: Das *Restnetzwerk* zu einem gegebenen Flussnetzwerk *und* einem Fluss besteht aus jenen Kanten, die mehr Fluss aufnehmen können. Ein *Erweiterungspfad* ist ein einfacher Pfad von s nach t im Restnetzwerk.
- Findet man in einem Restnetzwerk G_f (f ist ein Fluss in dem Flussnetzwerk G) einen weiteren Fluss f', so ist $f + f'$ erneut ein Fluss in G mit dem Wert $|f| + |f'| > |f|$.
- Letztere gilt insb. auch, wenn man über einen Erweiterungspfad so viel wie möglich 'leitet' (Restkapazität).

Die Ford-Fulkerson-Methode

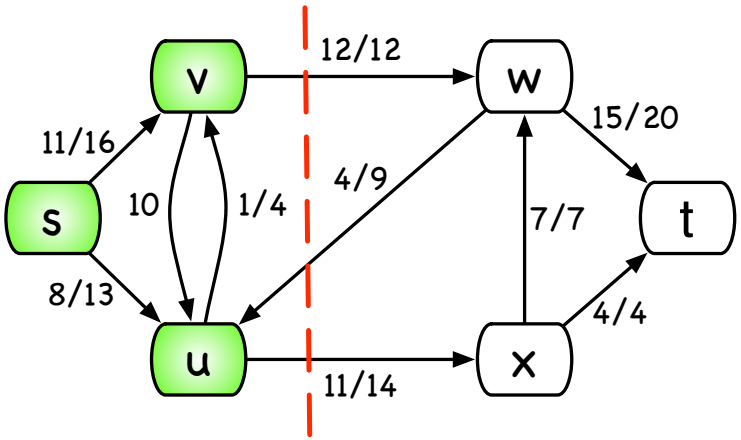
Die Ford-Fulkerson-Methode:

0. Sei $f = 0$.
1. Bestimme $G_f = (V, E_f)$ und c_f.
2. Bestimme einen Erweiterungspfad p. (Gibt es keinen, gehe zu 5.)
3. Erhöhe f entlang p um $c_f(p)$.
4. Gehe zu 1.
5. f ist ein maximaler Fluss.

Anmerkung

Die Idee: Bestimme so lange wie möglich Erweiterungspfade und erhöhe den Fluss jeweils um die Restkapazität. Laufzeit: $O(E \cdot |f^*|)$, wobei f^* ein maximaler Fluss ist.

Schnitte: Definition



Definition (Schnitt)

Ein *Schnitt* (S, T) eines Flussnetzwerkes $G = (V, E)$ ist eine Partionierung von V in S und $T = V - S$, wobei $s \in S$ und $t \in T$ gilt. Der *Nettofluss* und die *Kapazität* eines Schnittes (S, T) ist durch $f(S, T)$ bzw. $c(S, T)$ definiert. Ein *minimaler Schnitt* ist ein Schnitt, dessen Kapazität die kleinste von allen Schnitten des Netzwerkes ist.

Max-Flow-Min-Cut-Theorem

Satz

Sei $G = (V, E)$ mit c, s, t ein Flussnetzwerk, f ein Fluss in G. Die folgenden Bedingungen sind äquivalent:

1. *f ist ein maximaler Fluss in G.*
2. *Das Restnetzwerk G_f enthält keine Erweiterungspfade.*
3. *Es ist $|f| = c(S, T)$ für einen Schnitt (S, T) von G.*

Anmerkung

Mit diesem Satz folgt die Korrektheit der Ford-Fulkerson-Methode

Edmonds-Karp-Algorithmus

Der Edmonds-Karp-Algorithmus:

- Ermittle einen Erweiterungspfad p in der Ford-Fulkerson-Methode durch eine Breitensuche, d.h. p ist ein *kürzester Pfad* von s nach t im Restnetzwerk (wobei jede Kante das Gewicht 1 hat).

Satz

Die Anzahl der durch den Edmonds-Karp-Algorithmus vorgenommenen Flusserweiterungen ist in $O(V \cdot E)$. Da jede Iteration (der Prozedur FordFulkerson) in Zeit $O(E)$ implementiert werden kann, ist der Edmonds-Karp-Algorithmus in $O(V \cdot E^2)$.

Zusammenfassung: Graphen

- Grundlagen
  - Definition und Darstellung
  - Tiefen- und Breitensuche
- Topologisches Sortieren
- Starke Zusammenhangskomponenten
- Minimale Spannbäume
  - Definition
  - Algorithmus von Kruskal
  - Algorithmus von Prim
- Bestimmen der kürzesten Pfade
  - Definition und Varianten
  - (Breitensuche)
  - Algorithmus von Dijkstra
  - Algorithmus von Bellman-Ford
  - Algorithmus von Floyd-Warshall
- Flüsse (Ford-Fulkerson-Methode/Edmonds-Karp-Algorithmus)

Komplexitätstheorie

Zusammenfassung:

Komplexitätstheorie (P, NP, ...)

Akzeptieren und entscheiden

Definition (Entscheiden von Sprachen)

Eine Sprache L wird von einem Algorithmus A *entschieden*, wenn

- jedes $x \in L$ von A akzeptiert wird ($A(x) = 1$) und
- jedes $x \notin L$ von A zurückgewiesen wird ($A(x) = 0$).

A hält also insb. auf jeder Eingabe an und liefert stets das richtige Ergebnis.

Anmerkung

- Beim *akzeptieren* von Sprachen wird die zweite Eigenschaft nicht gefordert.
- $x \in L$ heisst einfach nur, dass x eine Instanz ist, die wir akzeptieren wollen, also z.B. $x = (5, 5, 10)$ beim Summationsproblem (bzw. $x = \langle(5, 5, 10)\rangle$).

Die Klasse P

Definition (Polynomielle Zeitschranken)

- Ein Algorithmus A akzeptiert eine Sprache L in *polynomialer Zeit*, wenn sie von A akzeptiert wird und zusätzlich ein $k \in \mathbb{N}$ existiert, so dass jedes $x \in L$ mit $|x| = n$ in Zeit $O(n^k)$ akzeptiert wird.
- Soll A die Sprache L entscheiden, wird für jeden String x der Länge n in Zeit $O(n^k)$ entschieden, ob $x \in L$ gilt (oder nicht).

Definition (Defintion von P)

$$P = \{L \subseteq \{0,1\}^* \mid \begin{array}{l}\text{es existiert ein Algorithmus } A\text{, der } L \\ \text{in Polynomialzeit entscheidet.}\end{array} \}$$

Verifikation in polynomialer Zeit

Definition (Verifikationsalgorithmus)

Ein *Verifikationsalgorithmus* A ist ein Algorithmus mit zwei Argumenten $x, y \in \Sigma^*$, wobei x die gewöhnliche Eingabe und y ein *Zertifikat* ist. A *verifiziert* x, wenn es ein Zertifikat y gibt mit $A(x, y) = 1$. Die von A verifizierte Sprache ist

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : A(x, y) = 1\}.$$

Anmerkung

Es geht also insb. um die Eingabe x. Diese bilden die Sprache. Das Zertifikat y kann vom Algorithmus genutzt werden, um zu entscheiden, ob $x \in L$ gilt, oder nicht.

Die Klasse *NP*

In *NP* sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (*NP*)

$L \in NP$ gdw. ein Algorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c

$$L = \{x \in \{0,1\}^* \mid \begin{array}{l}\text{es existiert ein Zertifikat } y \text{ mit } |y| \in O(|x|^c), \\ \text{so dass } A(x,y) = 1 \text{ gilt}\end{array} \}$$

gilt.

Sätze

Satz

Für P ist akzeptieren und entscheiden identisch, d.h. die gleiche Sprachfamilie.

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

Satz

Sind $L_1, L_2 \in P$, so gilt auch $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, \overline{L_1}, L_1^* \in P$.

Satz

Sind $L_1, L_2 \in NP$, so gilt auch $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, L_1^* \in NP$.

Vier Möglichkeiten

Definition

$L \in coNP$ gdw. $\overline{L} \in NP$

Wir wissen weder ob $P = NP$, noch ob $NP = coNP$ gilt. Eine der folgenden vier Möglichkeiten gilt:

- $P = NP = coNP$
- $P \subset NP = coNP$
- $P = NP \cap coNP \subset coNP$ und $P = NP \cap coNP \subset NP$ und $NP \neq coNP$
- $P \subset NP \cap coNP \subset coNP$ und $P \subset NP \cap coNP \subset NP$ und $NP \neq coNP$

Wir haben leider keine Ahnung welche. Die vierte ist am wahrscheinlichsten... *Tipp: Alle Möglichkeiten mal aufmalen!*

Reduktion - Wiederholung

Definition (Reduktion)

Seien $L_1, L_2 \subseteq \{0,1\}^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2 *in polynomialer Zeit reduziert wird*, wenn eine in Polynomialzeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert mit

$$x \in L_1 \text{ genau dann wenn } f(x) \in L_2$$

für alle $x \in \{0,1\}^*$ gilt. Hierfür schreiben wir dann $L_1 \leq_p L_2$. f wird als Reduktionsfunktion, ein Algortihmus der f berechnet als Reduktionsalgorithmus bezeichnet.

NP-vollständig - Wiederholung

Definition

Eine Sprache $L \subseteq \{0,1\}^*$ wird als NP-vollständig bezeichnet, wenn

1. $L \in NP$ und
2. $L' \leq_p L$ für jedes $L' \in NP$ gilt.

Kann man für L zunächst nur die zweite Eigenschaft beweisen, so ist L NP-schwierig (-schwer/-hart).
Alle NP-vollständigen Probleme bilden die Komplexitätsklasse NPC.

Reduktion - Sätze

Satz (Lemma 34.3 im (Cormen))

Seien $L_1, L_2 \subseteq \{0,1\}^*$ mit $L_1 \leq_p L_2$, dann folgt aus $L_2 \in P$ auch $L_1 \in P$.

Satz

Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq L_3$.

Theorem (Theorem 34.4 im (Cormen))

Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.

Theorem

Sei L eine Sprache und $L' \in NPC$. Gilt $L' \leq_p L$, so ist L NP-schwierig. Ist zusätzlich $L \in NP$, so ist L NP-vollständig.

Verfahren

Methode zum Beweis der NP-Vollständigkeit einer Sprache L:

1. Zeige $L \in NP$.
2. Wähle ein $L' \in NPC$ aus.
3. Gib einen Algorithmus an, der ein f berechnet, das jede Instanz $x \in \{0,1\}^*$ von L' auf eine Instanz $f(x)$ von L abbildet.
4. Beweise, dass f die Eigenschaft $x \in L'$ gdw. $f(x) \in L$ für jedes $x \in \{0,1\}^*$ besitzt.
5. Beweise, dass f in Polynomialzeit berechnet werden kann.

Anmerkung

Die letzten drei Punkte zeigen $L' \leq_p L$. Mit dem vorherigen Satz folgt daraus und aus den ersten beiden Punkten $L \in NPC$.

NP-vollständige Probleme

Definition (SAT)

$\texttt{SAT} = \{\langle \phi \rangle \mid \phi \text{ ist eine erfüllbare aussagenlogische Formel}\}$

Definition (CNF)

$\texttt{CNF} = \{\langle \phi \rangle \mid \phi \text{ ist eine erfüllbare aussagenlogische Formel in KNF}\}$

Definition (3CNF)

$\texttt{3CNF} = \{\langle \phi \rangle \mid \phi \in$
CNF und jede Klausel hat genau drei verschiedene Literale$\}$

Definition (Clique)

$\texttt{CLIQUE} = \{\langle G, k \rangle \mid G$ enthält einen K^k als Teilgraphen

...

Die Grenzen des Machbaren...

NP-schwierige Probleme sind aller Wahrscheinlichkeit nach *nicht effizient lösbar*. Die meisten dieser Probleme sind aber wichtig und treten in vielen Anwendungen (in Verkleidung) auf. Wie lösen wir die also trotzdem?

Die Grenzen und darüberhinaus

Möglichkeiten (auch für den normalen Algorithmenentwurf):

- die betrachteten Strukturen einschränken
- Approximationsverfahren
- Randomisierte Algorithmen
- Heuristiken
- Fixed-Parameter-Algorithmen
- ...

Das Wechselspiel

Man entwickelt hier dann neue (Algorithmen-)Techniken und sucht (und findet) neue Grenzen - um die man dann wieder versucht herumzukommen ...

Die Herausforderung

Das Wechselspiel zwischen Komplexitätstheorie und Algorithmik kann man beschreiben als den Versuch *die Grenzen des (praktisch) möglichen zu verstehen und diese Grenzen immer weiter hinauszuschieben*.

Ein bisschen Werbung

Werbung

Im kommenden Sommersemester:

- Titel: Lösungsstrategien fur NP-schwere Probleme der Kombinatorischen Optimierung
- Lehrender: Prof. Dr. Thomas Andreae
- Semesterwochenstunden: 2 (+ 2 SWS Übungen)

Z.B. im Wahlbereich möglich. Teilnahme möglich, wenn AuD oder Optimierung erfolgreich abgeschlossen wurde.

Letzter Hinweis

Ein letzter Hinweis

Wenn man mehr Lernen möchte: Die Vorlesung noch einmal wiederholen (und mit Literatur ergänzen) und dabei auf Algorithmen*techniken* achten:

- Greedy-Verfahren
- Divide & Conquer
- Dynamisches Programmieren
- Backtracking
- Branch & Bound
- ...

Viel Spass und Erfolg!

Ende