Info I-Zusammenfassung

1 Grundlagen

Definition 1.1 Algorithmus

Ein Algorithmus wird gekennzeichnet durch die folgenden Eigenschaften:

- Schrittweise Ausführung
- Determiniertheit
- Eindeutigkeit
- Jeder Schritt bewirkt eine Änderung
- Endlichkeit jedes einzelnen Schrittes, aber nicht unbedingt des Algorithmus
- Endlichkeit der Beschreibung

Definition 1.2 Von-Neumann-Rechner

Ein *Von-Neumann-Rechner* ist ein speicherprogrammierter Rechner.

2 Grammatiken

Definition 2.1 Alphabet

Ein Alphabet V ist eine endliche Menge von Zeichen.

Definition 2.2 Zeichenfolge

Eine Zeichenfolge f ist eine endliche, geordnete Menge von Zeichen aus dem Alphabet. ε bezeichnet man die leere Zeichenkette.

Schreibweise: $a^n := aa^{n-1}, a^0 := \varepsilon, a\varepsilon = a$

$x \in V, a$ Zeichenfolge : $|xa| := |a| + 1, |\varepsilon| := 0$

V^* bezeichnet die Menge der Zeichenfolgen über V, einschließlich ε.

Definition 2.3 Grammatik

Eine *Grammatik* ist ein Quadrupel von Mengen $G = (V_T, V_N, P, S)$. Dabei nennt man V_T die Menge der terminalen Symbole, V_N die Menge der nicht-terminalen Symbole, P die Menge der Produktionen und S das Startsymbol.

Es muss gelten: $V_T \cap V_N = \emptyset$, $S \in V_N$. Weiterhin bezeichnet man mit $V = V_T \cup V_N$ das Vokabular der Grammatik.

Für alle Elemente $p \in P$ gilt: $p = l \to r$ mit zwei Zeichenfolgen l und r, für die gilt: $l = a\alpha b$ mit $a, b \in V^*, \alpha \in V_N$, $r \in V^*$.

Definition 2.4 Ableitung

Eine Zeichenfolge β ist eine *direkte Ableitung* einer Zeichenkette α ($\alpha \to \beta$), wenn $\alpha = w_1 l w_2$, $\beta = w_1 r w_2$ und $l \to r$ eine der Produktionen der Grammatik ist.

Eine *Ableitung* ist dann eine beliebige (endliche) Aneinaderreihung von direkten Ableitungen.

Definition 2.5 Satzform/Phrase

Eine Zeichenfolge α ist eine *Satzform/Phrase* einer Grammatik, wenn α eine Ableitung des Startsymbols ist.

Definition 2.6 Sprache

Die Sprache $L(G)$ einer Grammatik G besteht aus der Menge der Phrasen der Grammatik, die nur aus Terminalsymbolen bestehen.

Definition 2.7 Chomsky-Klasse

Eine Grammatik gehört einer der folgenden *Chomsky-Klassen* an, falls die entsprechende Bedingung in der Tabelle zutrifft.

Chomsky-Klasse	**Produktion**	**Bedingung**
0 ("unbeschränkt")	$l \to r$	$l, r \in V^*$
1 ("beschränkt")	$l \to r$	$l, r \in V^*, 1 \leq \lvert l \rvert \leq \lvert r \rvert$
1 ("kontextsensitiv")	$ulv \to urv$	$l \in V_N, u, r, v \in V^*, r \neq \varepsilon$
2 ("kontextfrei")	$l \to r$	$l \in V_N, r \in V^*$
3 ("linkslinear","regulär")	$\frac{l \to r}{l \to ur}$	$l, u \in V_N, r \in V_T^*$
3 ("rechtslinear","regulär")	$\frac{l \to r}{l \to ru}$	$l, u \in V_N, r \in V_T^*$

Bei C_1-Grammatiken ist zusätzlich $S \to \varepsilon$ eine erlaubte Produktion.(Das Startsymbol S darf dann allerdings in keiner Expansion einer Produktion vorkommen.)

Bei C_2-Grammatiken ist es möglich, Produktionen der Form $l \to \varepsilon$ zu eliminieren.

Es gilt: $C_3 \subseteq C_2 \subseteq C_1 \subseteq C_0$. Zur Zerteilung von C_2- und C_3-Sprachen gibt es effiziente Algorithmen.

3 Boole'sche Algebra

Definition 3.1 Boole'sche Algebra

Eine Menge B mit zwei Verknüpfungen $\wedge$ und $\vee$ heißt *Boole'sche Algebra*, falls die folgenden Axiome gelten:

(1) $(\forall a, b \in B)(b \vee a = a \vee b)$ (*Kommutativität* $\vee$)

(2) $(\forall a, b \in B)(b \wedge a = a \wedge b)$ (*Kommutativität* $\wedge$)

(3) $(\forall a, b, c \in B)(a \vee (b \vee c) = (a \vee b) \vee c)$ (*Assoziativität* $\vee$)

(4) $(\forall a, b, c \in B)(a \wedge (b \wedge c) = (a \wedge b) \wedge c)$ (*Assoziativität* $\wedge$)

(5) $(\forall a, b \in B)(a \wedge (a \vee b) = a)$ (*Verschmelzung* $\vee$)

(6) $(\forall a, b \in B)(a \vee (a \wedge b) = a)$ (*Verschmelzung* $\wedge$)

(7) $(\forall a, b, c \in B)(a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c))$ (*Distributivität* $\vee$)

(8) $(\forall a, b, c \in B)(a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c))$ (*Distributivität* $\wedge$)

(9) $(\forall a \in B)(a \vee 0 = a)$ (*Neutralelement* $\vee$)

(10) $(\forall a \in B)(a \wedge 1 = a)$ (*Neutralelement* $\wedge$)

(11) $(\forall a \in B)(a \vee (\neg a) = a)$ (*Inverses* $\vee$)

(12) $(\forall a \in B)(a \wedge (\neg a) = 0)$ (*Inverses* $\wedge$)

Definition 3.2 Implikation/Äquivalenz/Antivalenz

(1) $a \implies b :\Leftrightarrow \neg a \vee b$ (*Implikation*)

(2) $a \Leftrightarrow b :\Leftrightarrow (\neg a \wedge \neg b) \vee (a \wedge b)$ (*Äquivalenz*)

(3) $a \Leftrightarrow b :\Leftrightarrow (a \wedge \neg b) \vee (\neg a \wedge b)$ (*Antivalenz*)

Satz 3.1 Weitere Regeln

(1) $a = \neg\neg a$ (*Doppelte Negierung*)

(2) $a \wedge a = a$ (*Idempotenz* $\wedge$)

(3) $a \vee a = a$ (*Idempotenz* $\vee$)

(4) $\neg(a \vee b) = \neg a \wedge \neg b$ (*DeMorgan'sche Regel* $\vee$)

(5) $\neg(a \wedge b) = \neg a \vee \neg b$ (*DeMorgan'sche Regel* $\wedge$)

(6) $(a \implies b) \wedge (b \implies c) \implies a \implies c$ (*Transitivität*)

Definition 3.3 Tautologie

Eine *Tautologie* ist eine Aussage, die immer zutrifft.

4 Induktion

Verschiedene Induktionsverfahren (z.z. Aussage: $A(n) \forall n$)

- Zeige: $A(1)$, $A(n) \implies A(n+1)$
- Zeige: $A(1)$, $A(n-1) \implies A(n)$
- Zeige: $A(1)$, $(\forall k \in \mathbb{N})(k < n \implies A(k)) \implies A(n)$ ("starke" Induktion)
- Zeige: $A(1), A(2)$, $A(n-2) \implies A(n)$
- Zeige: $A(1)$, $A(\frac{n}{2}) \implies A(n)$ (Aussage gilt dann aber nur für $n = 2^k$)
- Sei $M_A \subseteq \mathbb{N}$ unendlich. Zeige $(\forall n \in M_A)(A(n))$ und $A(n) \implies A(n-1)$

Tricks bei Induktionsbeweisen:

- Aussage zunächst nicht über Ausdruck selbst, sondern nur über sein Wachstum
- Keine Eile! Versuche evtl. verschachtelte Induktionen. Teile eine schwierige Behauptung in mehrere einfache Teile!
- Hole alles aus der Induktionsvoraussetzung heraus! (Benutze sie evtl. mehrfach in versch. Situationen)
- Induktion über mehrere Parameter: Wichtig ist eine geschickte Wahl des Induktionsparameters (vielleicht auch eine Kombination?)
- Benutze evtl. starke Induktion, zeige nur, daß die Behauptung aus irgendeinem der vorhergehenden Fälle folgt.
- Zwei Fälle: Lohnt es sich, beide gemeinsam zu betrachten, so daß sie beide aufeinander aufbauen?
- Kann die Induktionsvoraussetzung verstärkt werden (so daß ein stärkerer Beweis entsteht, der dennoch einfacher ist)?
- Gilt der Schluß für *wirklich* alle n?
- Falls anwendbar: Beziehe dich vielleicht nicht auf die "ersten", sondern auf die "letzten" Elemente der Behauptung.

Definition 4.1 Schleifeninvariante

Eine Bedingung, die zu Schleifenbeginn, nach jeder Ausführung der Schleife und an deren Ende wahr ist, nennt sich *Schleifeninvariante*.

5 O-Kalkül

Definition 5.1 O-Kalkül

Seien $f(n), g(n) : \mathbb{N} \to \mathbb{N}$ zwei Funktionen. Dann gilt:

$$f(n) = O(g(n)) :\Leftrightarrow (\exists n_0, c \in \mathbb{N})(\forall n > n_0)(\frac{f(n)}{g(n)} \leq c)$$
$$f(n) = \Omega(g(n)) :\Leftrightarrow (\exists n_0, c \in \mathbb{N})(\forall n > n_0)(\frac{f(n)}{g(n)} \geq c)$$
$$f(n) = \Theta(g(n)) :\Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$
$$f(n) = o(g(n)) :\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) = \omega(g(n)) :\Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \Leftrightarrow g(n) = o(f(n))$$

Satz 5.1

Voraussetzung: $c > 0, a > 1, f(n) \nearrow$

$$f^c(n) = O(a^{f(n)})$$
$$f^c(n) = o(a^{f(n)})$$

Satz 5.2

Voraussetzung: $f(n) = O(s_f(n)), g(n) = O(s_g(n))$

$$f(n) + g(n) = O(s_f(n) + s_g(n))$$
$$f(n) \cdot g(n) = O(s_f(n) \cdot s_g(n))$$

6 Rekurrenzrelationen

Tricks für Rekurrenzen:

- Sei $T(g(n)) = E(T(n), n)$ eine gegebene Rekurrenz. Wir wollen zeigen, daß $T(n) \leq f(n)$. Dann muß gezeigt werden:

$$f(g(n)) \geq E(f(n), n)$$

Tricks zum Finden von geschlossenen Ausdrücken:

- Anfang hinschreiben (ausmultipliziert, nicht ausmultipliziert, faktorisiert, ...)
- Summenindex verschieben, voneinander abziehen, dann z.B. Differenz wie $2T(n) - T(n)$ betrachten.
- Nimm vielleicht zunächst an, daß n eine bestimmte Form hat: z.B. $n = 2^k$
- Rekurrenzen: Gibt es eine charakteristische Gleichung? (in der Art von $a_{n+1}^2 = a_n + c$)?
- Wachstum betrachten: Was sagt $T(n+1) - T(n)$?
- "Full-history"? Eliminiere die Geschichte mittels geschickter Subtraktion.

Beispiel "Teile-und-Herrsche"-Relationen

Ein rekursiver Algorithmus zerteile bei der Bearbeitung eines Problems der Größe n das Problem in a Teilprobleme der Größe $\frac{1}{b}$. Die Kombination der Lösungen der Teilprobleme zu der des Gesamtproblems benötige $c \cdot n^k$. Damit gilt:

$$T(n) = aT\left(\frac{n}{b}\right) + c \cdot n^k$$

Als Abschätzung hierfür kann verwendet werden:

$$T(n) = \begin{cases} O(n^{\log_b a)} & \text{falls } a > b^k \\ O(n^k \log n) & \text{falls } a = b^k \\ O(n^k) & \text{falls } a < b^k \end{cases}$$

7 Datenstrukturen

Definition 7.1 Abstrakter Datentyp

Template auf einen Typ, gekoppelt mit bestimmten Anforderungen, z.B. Wohlordnung

Definition 7.2 Multiset

Eine ungeordnete Struktur, die mehrere Vorkommen eines einzelnen Elements darstellen kann.

Definition 7.3 Heap

Baum, in dem Eltern-Elemente stets größer als ihre Kind-Elemente sind.

Einfügen: Element irgendwo am Ende einfügen und dann so lange tauschen, bis Heap-Eigenschaft wiederhergestellt ist.

L"schen der Wurzel: Wurzel weg, dann irgendein Blatt an der Wurzel einfügen und dann so lange tauschen, bis Heap-Eigenschaft wiederhergestellt ist.

Definition 7.4 Dictionary

Eine abstrakte Datenstruktur, die effizient einfügen, löschen und suchen kann.

Definition 7.5 Binary Search Tree/BST

Suchen/Einfügen: Klar

Löschen: (nur schwierig, fallls 2 Kinder vorhanden, in diesem Fall) Vorgänger/Nachfolger suchen, tauschen (Baum nicht mehr konsistent), Element hat jetzt nur max. ein Kind (Def. Vorgänger/Nachfolger!), Löschen, Baum wieder konsistent.

Definition 7.6 AVL Tree/Balanced BST

Eine Seite eines (Unter-)Baumes ist höchstens 1 Element höher als die andere.

Einfügen: Eine (einfache oder doppelte) "Rotation" genügt, um die AVL-Eigenschaft beizubehalten. Zentraler Punkt: "Kritischer Knoten": Wurzel des kleinsten Baumes, der aus dem Gleichgewicht gerät. Hier findet die Rotation statt.

Löschen: Offensichtlich ein bißchen komplizierter. :-)

Definition 7.7 Hashing

BLAH.

Sei p eine Primzahl, m die Größe der Hashtabelle, a, b zwei Zufallszahlen, x der Schlüssel als Integer. Dann gelten gemeinhin als "gute" Hash-Funktionen:

$$f_1(x) = x \bmod m$$
$$f_2(x) = (x \bmod p) \bmod m$$
$$f_3(x) = ((ax + b) \bmod p) \bmod m$$

Bemerkung Repräsentation von Graphen

Ein Graph $G = (V, E)$ kann mindestens auf diese Arten und Weisen datentechnisch repräsentiert werden:

- "Nachbarschaftsmatrix"
- Zeiger-Netz
- $|V|+|E|$-Array (nicht-dynamisch) mit Knoten zuerst, die Indizes in Array speichern, ab denen benachbarte Knoten indiziert werden.

8 Strategien für gute Algorithmen

- Versuche, das Problem zu reduzieren. Wende dazu folgende Techniken an:
  - Vielleicht lassen sich Teile des Problems einfach weglassen, ohne die Problemstellung zu ändern.

    Beispiel: Prominentenproblem (eine Person lässt sich mit nur einer Frage eliminieren), Diskrete Bijektion (auf Surjektivität spielen, nicht auf Injektivität)
  - Teile & Herrsche$^{\text{TM}}$ (mehrere gleiche kleinere Probleme)

    Beispiel: Skyline-Problem
  - Suche nach mehrere verschiedene, aber kleinere Teilproblemen, und wende diese Verfahren an
- Verstärkung der Induktionshypothese (Mach' einfach mehr, als du eigentlich machen musst)

  Beispiel: Maximale zusammenhängende Unterfolge (zusätzlich noch größtes Suffix liefern), Gleichgewichtsfaktoren in BSTs (zusätzlich noch Höhe berechnen)
- Dynamisches Programmieren Existiert zur Lösung des Problems eine Funktion $F(x, y)$ mit einigen (wenigen) Parametern, so lässt sich ein mittels Rekursion exponentiell lösbares Problem auch in $O(n^k)$ Schritten lösen. Es muss dabei eine Abhängigkeit des Wertes $F(x, y)$ von $F(x', y'), x' < x \vee y' < y$ geben.

  Bsp.: Rucksackproblem. Zugehörige Funktion $F(R)$ ergibt, ob ein Rucksack der Größe R mit bisherigen Elementen schon einmal erfolgreich bepackt wurde. Sei g die Größe des nächsten einzupackenden Elementes. Dann ist $F(R + g) = F(R)$ für alle R.

9 Algorithmen für Folgen und Mengen

Definition 9.1 Prinzip der Binärsuche

Gegeben sei eine sortierte (!) Folge oder Sequenz.

a) *Suche eines bestimmten Elements in einer bekannten Anzahl Elemente*: Man eliminiert jeweils die Hälfte des Suchraums, indem man das Element "in der Mitte" mit dem gesuchten vergleicht.

b) *Suche eines bestimmten Elements in einer unbekannten Anzahl Elemente*: Man verdoppelt solange die angenommene Größe des Suchraums, bis ein Element an dessen hinteren Ende liegt, das größer als das gesuchte ist.

Dieses Prinzip lässt sich auch auf allgemeine "suche ein i, ab dem..." oder "suche ein i, bis zu dem..."-Probleme anwenden. Vergleiche auch: Intervallhalbierungsverfahren zur Bestimmung von Nullstellen

Zur Verfeinerung des Verfahrens kann man mit Hilfe einer Art "Tangentenbestimmung" einen günstigeren Teilungsort als immer die Hälfte bestimmen (vgl. Newton-Verfahren). ["Interpolationssuche"]

Bekannte Sortieralgorithmen:

Bucket sort Für jeden möglichen Schlüssel einen Bucket bereitstellen, einsortieren, zusammensammeln, fertig: $O(n)$

Radix sort Eine Art "rekursiver Bucket sort". Man stellt jeweils nur eine feste Anzahl Buckets bereit, sortiert ein, sortiert dann den Inhalt eines Buckets wieder rekursiv in eine feste Anzahl ein usw. $O(n)$

Straight radix sort Radix sort, bei dem die "unwichtigsten" (am differenzierendsten) Teile des Schlüssels zuerst betrachtet werden.

Radix-exchange sort Radix sort, bei dem die "wichtigsten" (am wenigsten differenzierenden) Teile des Schlüssels zuerst betrachtet werden.

Lexicographic sort (alphabetisches Sortieren mittels radix-exchange sort) ?

Insertion sort Elemente nacheinander an entsprechender Stelle einsortieren: $O(n^2)$

Selection sort Größtes Element auswählen, an letzte Stelle schieben, nächstkleineres auswählen, an vorletzte Stelle... usw. $O(n^2)$

Mergesort Teile Sequenz in zwei Hälften, sortiere rekursiv (bis zum Basisfall: 2 Elemente), kombiniere ("merge") dann die zwei sortierten Hälften zusammen. (nicht in-place) $O(n \log n)$

Quicksort Pivot wählen, Sequenz anhand dieses Pivots in zwei Hälften teilen, rekursiv wieder das gleiche für beide Hälften. In der Regel $O(n \log n)$, im schlimmsten Fall $O(n^2)$.

Heapsort Baue Heap nach einem der folgenden Verfahren auf:

top-down Betrachte die Sequenz bis einschließlich $i-1$ als Heap, füge das i-te Element ein (gemäß Heap-Einfüge-Algorithmus)

bottom-up "Unterste Zeile" des Baumes (bis einschl. $\lfloor n/2 \rfloor$) besteht bereits aus (Einer-)Heaps. Jetzt kann jedes potentielle neue (Eltern-)Element so weit im Baum nach unten wandern, bis das Heap-Kriterium wieder erfüllt ist.

Jetzt kann mit Hilfe des Heap-Lösche-Algorithmus immer das maximale Element aus dem Heap entnommen werden. $\Theta(n \log n)$

Man kann zeigen, dass kein vergleichsbasierter Sortieralgorithmus in weniger als $O(n \log n)$ Schritten zum Ziel kommen kann. (Entscheidender Trick: Entscheidungsbaum)

Definition 9.2 Median

Der Median einer Menge M reeller Zahlen ist diejenige Zahl m, für die zwei Mengen $M_< := \{x \in M \mid x < m\}$ und $M_> := \{x \in M \mid x > m\}$ und eine bijektive Abbildung zwischen diesen Mengen existieren.

Problem k-kleinstes Element finden

Partitionieren wie Quicksort. Dann ist bekannt, in welcher Hälfte das gesuchte Element liegt.

Definition 9.3 Huffman encoding

Füge alle Zeichen nach ihrer umgekehrten Häufigkeit in einen Heap ein. Entnimm die zwei seltensten. Füge statt ihrer ein neues "Pseudo"-Zeichen mit der Summe ihrer Häufigkeiten in den Heap ein.

Problem Effizientes Durchsuchen von Text

KMP Berechne, wie weit der gesuchte Text weitergeschoben werden kann, wenn ein Match bei Position i fehlgeschlagen ist.

Boyer-Moore Betrachte letzten Buchstaben. Berechne, wie weit verschoben werden kann, so daß dieser zum ersten Mal richtig im gesuchten String zu liegen kommt.

Problem Finden einer minimalen Anzahl an Bearbeitungsschritten

Lösung Dynamisches Programmieren. Zeichenfolge A soll in Zeichenfolge B umgebröselt werden. $F(i,j) :=$ Anzahl der Schritte, die es braucht,um $A[1..i]$ in $B[1..j]$ umzubröseln. Abhängigkeit je nach zulässigen Bearbeitungsschritten.

Definition 9.4 Stochastische Algorithmen

Einen *Monte-Carlo-Algorithmus* nennt man einen Algorithmus, der mit einer gewissen (hohen) Wahrscheinlichkeit ein richtiges Ergebnis liefert. (Beispiel: Element in der oberen Hälfte)

Einen *Las-Vegas-Algorithmus* nennt man einen Algorithmus, der zwar mit Sicherheit ein richtiges Ergebnis liefert, dessen Laufzeit aber unbestimmt ist.

Ein Zufallsgenerator: Sei $r(1), b, t$ gegeben. Dann ist $r(i) = (r(i-1) \cdot b + 1) \bmod t$ eine "ordentliche" Zufallszahl, wenn $t >> 10^6, b \approx \frac{t}{10}$.

Problem Finden einer absoluten Mehrheit

Beobachtung: Die Entfernung zweier ungleicher Stimmen ist mehrheitsneutral.

Problem Finden der längsten aufsteigenden Teilfolge

Speichere alle bisherigen besten Teilfolgen und update sie entsprechend dem gerade bearbeiteten Element

10 Algorithmen für Graphen

Definition 10.1 Graph

Ein *Graph* $G = (V, E)$ besteht aus einer Menge V von Knoten (vertices) und einer Menge E von Kanten (edges). Es gibt *gerichtete* und *ungerichtete* Graphen. Bei ersteren besteht die Menge V aus geordneten, bei letzterem aus ungeordneten Paaren.

Der *Grad* eines Knotens $v \in V$ ist die Anzahl an Kanten, die an v enden. (analog: Einwärts-Grad/Auswärts-Grad)

Ein *Pfad* ist eine Folge von Knoten $v_1, v_2, ..., v_n$, für die eine Folge von Kanten $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$ existiert. Ein Pfad heißt *einfach*, falls kein Knoten in ihm zweimal vorkommt.

Ein Graph heißt *verbunden*, wenn zwischen allen Knoten Pfade existieren. Ein Graph heißt *zweifach/n-fach verbunden* (biconnected/n-connected), wenn zwischen je zwei Knoten mindestens zwei/n knotendisjunkte Pfade existieren.

Ein Knoten v_2 heißt *erreichbar* von v_1, falls ein Pfad zwischen v_1 und v_2 existiert.

Sei $U \subseteq V$ eine Menge von Knoten. Dann heißt $H = (U, F)$ der *von U induzierte Subgraph*, wenn F alle zu U gehörigen Kanten beinhaltet.

Eine *unabhängige Menge S in G* ist eine Menge von Knoten, die paarweise nicht durch Kanten verbunden werden.

Ein *Kreis* in G ist ein Pfad P mit $|P| > 1$ und dem gleichen Anfangs- und Endpunkt, in dem kein Knoten doppelt vorkommt. (circuit: Kreis, cycle: einfacher Kreis)

Ein Graph heißt *Wald*, falls er keine Kreise enthält. Ein verbundener Wald heißt *Baum*. Ist ein Knoten in ihm besonders ausgezeichnet, so heißt der Graph ein *Baum mit Wurzel*.

Ein *aufspannender Wald* eines Graphen ist ein Wald, der alle Knoten des Graphen enthält. Analog: aufspannender Baum ("spanning tree")

Ein *bipartiter* Graph ist ein Graph, dessen Knoten in zwei Mengen aufgeteilt sind, so daß jeder Knoten mit einem Knoten aus der jeweils anderen Menge verbunden ist.

Ein Graph heißt *eulersch*, wenn er verbunden ist und seine Knoten von geradem Grad sind.

Fläche: Keine Definition gefunden.

Satz 10.1

Voraussetzung: $G = (V, E)$ ein Graph, F die Menge der Flächen von G

$$|V| + |F| = |E| + 2$$

Definition 10.2 Tiefensuche (DFS: Depth first search)

Gehe zu einem Knoten, markiere ihn, klappere rekursiv alle benachbarten Knoten ab, vermeide dabei bereits markierte Knoten.

Versieht man jeden Knoten in der Reihenfolge der Abarbeitung bei der DFS mit einer Nummer, so nennt man diese Nummer die *DFS-Nummer*.

Mittels DFS lässt sich aus einem ungerichteten Graphen in einem Anlauf ein DFS-Baum erzeugen, der folgenden Eigenschaft hat: Entweder ist eine Kante Teil des Baumes, oder sie ist eine Querkante ("cross edge"), d.h. sie verbindet im DFS-Baum einen Knoten mit einem seiner Vorgänger.

Ebenso lassen sich mittels DFS aus gerichteten Graphen DFS-Bäume erzeugen. (dafür sind gegebenenfalls im Gegensatz zu ungerichteten Graphen mehrere Anläufe nötig. Es kann ja schließlich mehrere "Hügel" im Graphen geben, von denen man hinunter, aber nicht mehr hinauf kommt.) Hier gibt es für eine Kante die folgenden Möglichkeiten:

- im Baum enthalten
- Querkante
- Vorwärtskante
- Rückwärtskante

Haupteigenschaft von gerichteten DFS-Bäumen ist, dass der DFS-Baum bezüglich der DFS-Numerierung ein Heap ist. (Eltern-Knoten hat kleinere Zahl als Kind-Knoten)

Definition 10.3 Breitensuche (BFS: Breadth first search)

Beginne bei einem Knoten. Markiere ihn, markiere alle seine Kinder und setze Verweise auf sie in einen FIFO. Nimm den nächsten so zu bearbeitenden Knoten aus dem FIFO.

Versieht man jeden Knoten in der Reihenfolge der Abarbeitung bei der BFS mit einer Nummer, so nennt man diese Nummer die *BFS-Nummer*.

Problem Alle kürzesten Pfade von einem Ausgangspunkt

(Kurz ist hier im Sinne von Gewichten an einem gewichteten Graphen zu verstehen) Gesucht wird jeweils die Länge des kürzesten Pfades von einem Ausgangspunkt zu allen Knoten eines Graphen.

Lösung: Jedem Knoten wird eine gegenwärtige Pfadlänge zugeordnet. (anfangs ∞) Dann werden die Knoten nacheinander bearbeitet und für dem bearbeiteten Knoten benachbarte Knoten überprüft, ob der Weg über den bearbeiteten Knoten kürzer als dessen gegenwärtiger ist.

Problem Aufspannender Baum mit minimalen Kosten

(Kosten sind hier wieder die Summe der Gewichte eines gewichteten Graphen)

Lösung: Einziger Unterschied zum "alle kürzesten Pfade"-Problem besteht darin, dass anfangs eine Kante und kein Knoten ausgewählt wird.

Problem Alle kürzesten Pfade zwischen allen Knoten

(Kurz ist hier im Sinne von Gewichten an einem gewichteten Graphen zu verstehen)

Lösung: Durchlaufe alle Knoten und ersetze Gewichte (∞ falls Kante nicht existent) zwischen ihnen durch Summe der Gewichte der Wege über den gerade durchlaufenen Knoten, falls diese kleiner als das bestehende Gewicht ist.

Definition 10.4 Transitive Hülle (transitive closure)

Die transitive Hülle $H = (V, F)$ eines Graphen $G = (V, E)$ besteht aus den gleichen Knoten und genau denjenigen Kanten zwischen zwei Knoten v_1 und v_2, für die ein Pfad in G von v_1 nach v_2 existiert.

Satz 10.2 Satz von Menger

Voraussetzung: $G = (V, E)$ ungerichtet, verbunden. $u, v \in V, (u, v) \notin E$

Die minimale Anzahl an zu entfernenden Knoten, die benötigt wird, um den Graph bezüglich u und v zu trennen, ist gleich der Maximalzahl an knotendisjunkten Pfaden zwischen u und v.

Satz 10.3 Satz von Whitney

Ein ungerichteter Graph ist k-verbunden $\Leftrightarrow$ k Knotenentfernungen erforderlich sind, um den Graph zu trennen.

Definition 10.5 Artikulationspunkt

Ein *Artikulationspunkt* eines Graphen ist ein Knoten, dessen Entfernung die Trennung des Graphen bewirkt.

Definition 10.6 n-fach verbundene Komponente

Eine n-fach verbundene Komponente eines Graphen ist die maximale Untermenge an Kanten, die einen n-fach verbundenen Graphen induzieren.

Satz 10.4

Zwei Kanten e und f gehören genau dann zur selben zweifach verbundenen

Komponente, wenn es einen einfachen Kreis gibt, der durch diese zwei Kanten führt.

Satz 10.5

Jede Kante gehört zu genau einer zweifach verbundenen Komponente.

Problem Finden von zweifach verbundenen Komponenten in Graphen

Lösung: Baue mittels DFS einen Baum auf, der für je eine zweifach verbundene Komponente einen Knoten enthält. Merke dir dazu die niedrigste DFS-Nummer, bis zu der ein gewisser Baumabschnitt wieder hinaufreicht. (über Rückwärtskanten, die ja von DFS bekanntlich ohnehin eliminiert werden) Der durch diese DFS-Nummer markierte Punkt ist ein Artikulationspunkt.

Definition 10.7 Stark verbundene Komponente

Eine stark verbundene Komponente ist eine maximale Untermenge eines gerichteten Graphen, so dass für je zwei Knoten v_1 und v_2 Pfade von v_1 nach v_2 und umgekehrt von v_2 nach v_1 existieren.

Satz 10.6

Zwei Knoten gehören zur selben stark verbundenen Komponente $\Leftrightarrow$ Es existiert ein Kreis, der beide Knoten enthält.