Karlsruher Institut für Technologie

Algorithmen 1 – Sommersemester 2018
Zuverlässige Softwaresysteme im Kontext der Automobilindustrie
`http://verialg.iti.kit.edu`

Prof. Dr. Carsten Sinz – Markus Iser

Klausur **04.09.2018**

A Gemischte Kleinaufgaben (20 Punkte)

A.1 Union-Find Datenstruktur (5 Punkte)

A.1.1 Oberflächlich betrachtet (1 Punkt)

Was verwaltet die Union-Find Datenstruktur? Nennen Sie einen Algorithmus, in dem diese Datenstruktur eingesetzt wird.

Antwort:

- Mit Union-Find werden Partionen von Mengen verwaltet.
- Beispiel: In Kruskal's Algorithmus zur Bestimmung eines MST.

A.1.2 Naive Implementierung (2 Punkte)

Erklären Sie die beiden Operationen `union` und `find` und die Grundidee hinter deren naiven Implementierung. Wie sind die worst-case Laufzeiten der beiden Operationen (im $\mathcal{O}$-Kalkül)?

Antwort:

- Zu jedem Element wird ein Verweis (auf ein Eltern-Element) abgespeichert, über welchen die Zughörigkeit zu einer Partition bestimmt werden kann. Das Ende der Verweiskette bestimmt den Repräsentanten.
- Wenn die Menge in einelementige Teilmengen paritioniert ist, ist jedes Element sein eigener Repräsentant. Oder: Rekursionsende mit parent[i]=i reicht auch.
- union: Vereinigen von Partitionen. Element wird Repräsentant eines anderen Elements: Laufzeit: $O(1)$ (bzw. $O(n)$, wenn die in union enthaltenen find-Schritte mitgerechnet werden).
- find: Um festzustellen, welcher Partition ein Element angehört. Über die Eltern-Relation wird der Repräsentant eines Elements gesucht: Laufzeit: $O(n)$.

A.1.3 Optimierte Implementierung (2 Punkte)

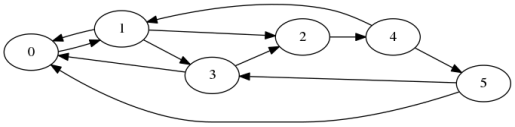
Was ist Union-Find mit *einfacher Pfadkompression* und welches Problem löst man mit der Kompression? Geben Sie die amortisierte Laufzeit der elementaren Operationen von Union-Find mit einfacher Pfadkompression im $\mathcal{O}$-Kalkül an.

Antwort:

- Pfade können sehr lang werden, daher worst-case O(n) bei find.
- Bei einfacher Pfadkompression werden in find alle Verweise auf dem Pfad direkt auf den obersten Repräsentanten "umgebogen".
- Dadurch hat find dann amortisierte Laufzeit von $O(\log n)$. union bleibt gleich.

A.2 Adjazenzfelddarstellung (2 Punkte)

Geben Sie den folgenden Graphen als Adjazenzfeld an.[1]



V	0	1	2	3	4	5					
	0	1	4	5	7	9					

E	0	1	2	3	4	5	6	7	8	9	10
	1	0	2	3	4	0	2	1	5	0	3

A.3 O-Kalkül (2 Punkte)

Zeigen oder widerlegen Sie, dass $n^{n+3} \in \mathcal{O}(n^n)$.

Antwort:
Angenommen $n^{n+3} \in \mathcal{O}(n^n)$, dann gilt:

$$\exists c \exists n_0 \forall n > n_0, n^{n+3} < c \cdot n^n$$

Aber:

$$n^{n+3} < c \cdot n^n \implies c > n^3$$

Widerspruch (c ist konstant, $n \to \infty$).

Alternatives Argument: $\lim_{n\to\infty} (\frac{n^{n+3}}{n^n}) \to \infty$. Ohne Begründung keinen Punkt.

A.4 Master-Theorem (3 Punkte)

Lösen Sie folgende Rekurrenzen im Θ-Kalkül mit $n = 7^k$ und $k \in \mathbb{N} \setminus \{0\}$.

$$X(n) = X\left(\frac{n}{7}\right) + 2018 \cdot n \qquad X(1) = 42 \tag{1}$$

$$Y(n) = 8 \cdot Y\left(\frac{n}{7}\right) + \frac{n}{2018} \qquad Y(1) = 6 \tag{2}$$

$$Z(n) = 7 \cdot Z\left(\frac{n}{7}\right) + X(n) \qquad Z(1) = 1 \tag{3}$$

Antwort:

1. $X \in \Theta(n)$
2. $Y \in \Theta(n^{\log_7 8}) \approx \Theta(n^{1.069})$
3. $Z \in \Theta(n \cdot \log n)$

A.5 Sortieren (3 Punkte)

A.5.1 Ordnen (1 Punkt)

Ordnen Sie die folgenden Algorithmen *absteigend*[2] nach ihrer worst-case Laufzeitkomplexität: Quicksort, Bucketsort, Mergesort.

Antwort:
Quicksort > Mergesort > Bucketsort.

[1] Das vorgegebene Raster ist als Hilfestellung gedacht, es geht *nicht* darum, unbedingt in jedes Feld etwas reinzuschreiben.
[2] Das heißt, die höchste Laufzeitkomplexität kommt in Ihrer Ordnung zuerst.

A.5.2 Untere Schranke (1 Punkte)

Wie lautet die untere Schranke für die worst-case Laufzeitkomplexität bei vergleichsbasiertem Sortieren? Gilt diese untere Schranke auch für Sortieralgorithmen für ganze Zahlen? Begründen Sie Ihre Antwort.

Antwort:

- Alle unteren Schranken: $O(n \cdot \log n)$ okay
- Tighte untere Schranken: $\Theta(n \cdot \log n)$ auch okay
- Mögliche Anzahl Vergleiche: $\Omega(n \cdot \log n)$, okay, kein Abzug für unpräzise Formulierung

Bei ganzen Zahlen gilt $O(n)$. Wegen der Struktur ganzer Zahlen kann man ziffernweise sortieren.

A.5.3 Stabiles Sortieren (1 Punkt)

Was macht ein *stabiles* Sortierverfahren aus? Nennen Sie ein Beispiel für ein stabiles Sortierverfahren.

Antwort:
Die relative Position gleich großer Elemente zueinander bleibt unverändert.
Beispiel: Mergesort, Insertionsort, (LSD)RadixSort, Bubblesort, k-Sort Prozedur (aus Vorlesung), Bucketsort

A.6 Dijkstras Algorithmus (1 Punkt)

In welchem Fall ist Dijkstras Algorithmus auf Graphen nicht anwendbar und warum?

Antwort:
Dijkstras Algorithmus läuft bei negativen Kreisen in eine Endlosschleife.

A.7 Hashing (4 Punkte)

A.7.1 Perfekte Hashfunktion (1 Punkt)

Unter welcher Bedingung erhält eine Hashfunktion das Attribut *perfekt*?

Antwort:
$x_1 \neq x_2 \implies h(x_1) \neq h(x_2)$

A.7.2 Kollisionen (2 Punkte)

Nennen und erklären Sie kurz zwei Möglichkeiten, um mit Kollisionen beim Hashing umzugehen.

Antwort:

- **linear probing (lineare suche, geschlossen):** Die nächste freie Position im Array wählen. Beim Zugriff wird im Array ab der Position bis zum nächsten freien Feld gesucht. Unter Umständen muss beim Löschen viel repariert werden.
- **linked lists (offen):** Pro Stelle eine verkettete Liste, an die angehängt wird. Beim Zugriff wird die Liste durchsucht, beim Löschen aus der Liste gelöscht.

A.7.3 Universelle Hashfunktionen (1 Punkt)

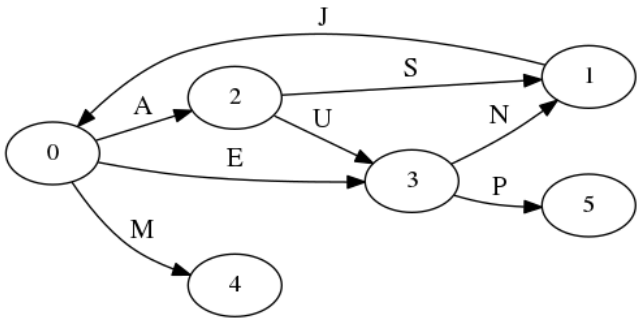
Welche Eigenschaften muss eine Familie X von Hashfunktionen haben, damit X *universell* ist?

Antwort:
Sei X Menge von Hashfunktionen $h : K \to \{1 \dots m\}$. Wenn für jedes Paar $k_0, k_1 \in K, k_0 \neq k_1$ die Anzahl der Hash-Funktionen $h \in X$, für die $h(k_0) = h(k_1)$ gilt, höchstens gleich $\frac{|X|}{m}$, dann gilt für eine zufällig aus X ausgewählten Hash-Funktion, dass die Kollisionswahrscheinlichkeit $\leq \frac{1}{m}$ ist. Damit ist X universelle Hashfamilie.

B Algorithmen Ausführen (20 Punkte)

B.1 Breitensuche (4 Punkte)

Führen Sie auf folgendem Graphen beginnend mit Knoten 0 eine Breitensuche aus. Halten Sie sich, wenn Sie die Wahl haben, in jeder Ebene an die von den Knotennamen induzierte Reihenfolge. Geben sie die Kanten in derjenigen Reihenfolge an, in der Sie sie bei der Breitensuche besucht haben, und benennen Sie, ob es sich um eine Tree-, Cross- oder Backward-Kante handelt.[3] Warum gibt es bei der Breitensuche keine Forward-Kanten?

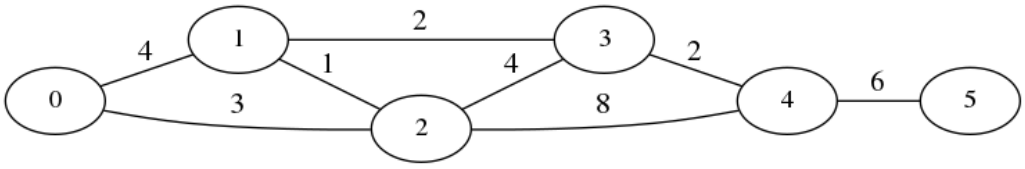


0	A	Tree	5	N	Cross
1	E	Tree	6	P	Tree
2	M	Tree	7	J	Backward
3	S	Tree	8		
4	U	Cross	9		

Es gibt keine Forward-Kanten bei der Breitensuche, weil diese direkt als Tree-Kanten besucht werden / ein Level übersprungen würde / etc.

B.2 Minimium Spanning Tree (3 Punkte)

Konstruieren Sie mit dem Algorithmus von Jarník und Prim für den unten dargestellten Graphen einen minimalen Spannbaum. Starten Sie bei Knoten 0. Zeichnen Sie den so berechneten minimalen Spannbaum noch einmal deutlich erkennbar darunter.



MST ist ein Pfad: $0 \to 2 \to 1 \to 3 \to 4 \to 5$

B.3 Dynamische Programmierung (3 Punkte)

Beim *Longest Common Subsequence Problem (LCS)* geht es darum, für zwei Zeichenketten S und T deren längste gemeinsame Teilfolge L zu berechnen. D.h. es ist die maximale Zeichenkette L gesucht, die sowohl in S als auch in T auftritt. L muss in S und T nicht zusammenhängend sein, aber die Reihenfolge der Zeichen muss beibehalten werden.

Beispiel: Für $S =$ `abazdc` und $T =$ `bacbad` ist $L =$ `abad` die gesuchte Lösung.

[3] Das vorgegebene Raster ist als Hilfestellung gedacht, es geht *nicht* darum, unbedingt in jedes Feld etwas reinzuschreiben.

Das LCS-Problem lässt sich mittels dynamischer Programmierung lösen. Dabei werden Teilprobleme für Präfixe $S[1..i]$, $i \leq |S|$ und $T[1..j]$, $j \leq |T|$ von S und T betrachtet. Wenn man die *Länge des LCS* der Präfixe $S[1..i]$ und $T[1..j]$ mit $\mathrm{LCS}(i,j)$ bezeichnet, so ergibt sich $\mathrm{LCS}(i,j)$ aus kleineren Teilproblemen wie folgt:

$$\mathrm{LCS}(i,j) = \begin{cases} \max(\mathrm{LCS}(i-1,j), \mathrm{LCS}(i,j-1)) & \text{falls } S[i] \neq T[j], \\ 1 + \mathrm{LCS}(i-1,j-1) & \text{falls } S[i] = T[j]. \end{cases}$$

Dabei bezeichnet $S[i]$ das i-te Zeichen in Zeichenkette S. Wir nehmen an, dass $\mathrm{LCS}[0,0] = \mathrm{LCS}[i,0] = \mathrm{LCS}[0,j] = 0$ ist.

B.3.1 Ausführen (2 Punkte)

Berechnen Sie die Werte $\mathrm{LCS}(i,j)$ für $1 \leq i,j \leq 6$ für $S =$ `abazdc` und $T =$ `bacbad` und tragen Sie diese in die folgende Tabelle ein:

		T					
		b	a	c	b	a	d
S	a	0	1	1	1	1	1
	b	1	1	1	2	2	2
	a	1	2	2	2	3	3
	z	1	2	2	2	3	3
	d	1	2	2	2	3	4
	c	1	2	3	3	3	4

B.3.2 Bestimmung der längsten gemeinsamen Teilsequenz (1 Punkt)

Beschreiben Sie, wie sich anhand der Werte $\mathrm{LCS}(i,j)$ die längste gemeinsame Teilsequenz, d.h. die eigentliche Lösung des LCS-Problems, berechnen lässt.

Antwort:
Starte in der Tabelle rechts unten; dies ist das aktuelle Feld. Wenn der Wert links oder über dem aktuellen Feld gleich dem Wert des aktuellen Feldes ist, gehe zu diesem Feld. Wenn beide Werte kleiner sind, speichere das Zeichen des aktuellen Feldes (dieses ist gleich in S und T) und gehe diagonal nach links oben. Die gespeicherten Zeichen in umgekehrter Reihenfolge sind die Lösung des LCS-Problems.

Aus der Tabelle aus B.3.1 ergibt sich: daba. (Die Ergebnis-Zeichenkette war nicht verlangt.)

B.4 Hashing mit Linear Probing (3 Punkte)

Gegeben sei die Hashfunktion $h(v) = v \bmod 13$. Sei T eine Hashtabelle, die unter Verwendung von Funktion $h(v)$ und mit *linear probing* ganzzahlige Werte in einem Feld der Länge 13 speichert. Führen Sie die folgende Menge von Befehlen auf T aus, und geben Sie den Zustand der Hashtabelle (des Feldes) nach jedem Schritt an.

	0	1	2	3	4	5	6	7	8	9	10	11	12
`T.insert 24`												24	
`T.insert 13`	13											24	
`T.insert 19`	13						19					24	
`T.insert 26`	13	26					19					24	
`T.insert 1`	13	26	1				19					24	
`T.insert 2`	13	26	1	2			19					24	
`T.remove 1`	13	26	2				19					24	
`T.insert 12`	13	26	2				19					24	12
`T.remove 13`	26		2				19					24	12
`T.remove 24`	26		2				19						12

B.5 Gnome Sort (3 Punkte)

Algorithmus 1 : Gnome Sort

Input : Array a
Output : Array a (sorted in ascending order)

```
1 pos ← 0
2 while pos < length(a) do
3     if pos = 0 or a[pos] ≥ a[pos −1] then
4         pos ← pos +1
5     else
6         swap a[pos] and a[pos −1]
7         pos ← pos −1
```

B.5.1 Ausführen (1,5 Punkte)

Führen Sie Algorithmus 1 für $a = [7, 3, 5, 9, 8]$ aus und geben Sie a nach jeder swap-Operation an.

pos	a=				
1	3	7	5	9	8
2	3	5	7	9	8
4	3	5	7	8	9

B.5.2 Laufzeitverhalten (1,5 Punkte)

Geben Sie die worst-case Laufzeit von Algorithmus 1 im Θ-Kalkül an und begründen Sie ihre Antwort.

Antwort:
Worst-case tritt ein, wenn an jeder Position bis ganz nach vorne geswapped wird.

$$2 \cdot \sum_{i=0}^{n-1} i = (n-1) \cdot (n-2) \in O(n^2)$$

B.6 Binäre Heaps (4 Punkte)

Gegeben Sei ein Feld mit den Zahlen $7, 8, 13, 20, 6, 19, 35, 12$.

Erstellen Sie mit den oben genannten Zahlen zunächst einen binären Heap.[4] Führen Sie dann zweimal deleteMin aus. Dann fügen Sie erst 17 und dann 4 ein.

Tragen Sie nach jedem Schritt den Zustand des Feldes in die Tabelle ein.

Antwort:

	makeheap-Methode aus Vorlesung:							
	7	8	13	20	6	19	35	12
`makeheap`	6	7	13	12	8	19	35	20
`deleteMin`	7	8	13	12	20	19	35	
`deleteMin`	8	12	13	35	20	19		
`insert 17`	8	12	13	35	20	19	17	
`insert 4`	4	8	13	12	20	19	17	35

[4] Zur Klarstellung: Es geht in dieser Aufgabe um binäre min-Heaps, die in einem Array gespeichert werden.

C Algorithmenentwurf (20 Punkte)

C.1 Konvexe Hülle (7 Punkte)

Zur Ermittlung der konvexen Hülle von n Punkten im $\mathbb{R}^2$ betrachten wir einen Algorithmus (in der Literatur als `Graham-Scan` Algorithmus bekannt), der gegeben eine Punktmenge P eine *minimale* Liste von Punkten $K \subseteq P$ berechnet, welche die *konvexe Hülle* von P aufspannt. Zur Vereinfachung gehen wir davon aus, dass für alle Punkte $(x, y) \in P$ gilt, dass $x \geq 0$ und $y \geq 0$.[5]

Initialisierung

Die Laufzeit des `Graham-Scan` Algorithmus wird durch das Sortieren bestimmt. In einem Initialisierungsschritt werden die Punkte $p \in P$ aufsteigend nach dem Winkel $\angle\vec{p}\vec{x}$ zwischen Vektor $\vec{p}$ und Einheitsvektor $\vec{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ sortiert.[6]

Build Convex Hull

Anschließend wird im Schritt `buildConvexHull` die Liste $K \subseteq P$ von Punkten erzeugt, welche die konvexe Hülle von P aufspannen. Implementieren Sie nur den Schritt `buildConvexHull` (in Pseudocode), der ein nach den oben genannten Kriterien sortiertes Array P von n Punkten engegennimmt und in $\mathcal{O}(n)$ Rechenschritten die konvexe Hülle K erzeugt.

Wählen Sie geschickt eine geeignete Datenstruktur für K. Verwenden Sie Funktion d aus Gleichung 1, um die relative Position eines Punktes zu einer durch zwei Punkte aufgespannten Geraden zu bestimmen.

Zeigen Sie, dass Ihre Implementierung die Laufzeit $\mathcal{O}(n)$ nicht überschreitet.

$$d(A, B, C) = \begin{vmatrix} 1 & x_A & y_A \\ 1 & x_B & y_B \\ 1 & x_C & y_C \end{vmatrix} = (x_B - x_A)(y_C - y_A) - (x_C - x_A)(y_B - y_A)$$

$$d(A, B, C) < 0 \iff C \text{ liegt rechts von } \overrightarrow{AB}$$
$$d(A, B, C) > 0 \iff C \text{ liegt links von } \overrightarrow{AB}$$
$$d(A, B, C) = 0 \iff C \text{ liegt auf } \overrightarrow{AB}$$

Gleichung 1: Relative Position eines Punktes C bezüglich eines Vektors $\vec{AB}$

[5] Punkte $\begin{pmatrix} x \\ y \end{pmatrix}$ werden in P als Tupel (x, y) dargestellt. Tun Sie das auch in Ihrem Pseudocode.

[6] Bei gleichem Winkel werden die Punkte p aufsteigend nach der Länge $|\vec{p}|$ sortiert.

Algorithmus 2 : buildConvexHull

Input : Array P of n Tuples
Output : Stack S of n Tuples

```
1  Stack[n] S;
   // Regular Pass
2  for p ∈ P do
3      while len(S) > 1 do
4          a ← S[len(S) − 2];
5          c ← S[len(S) − 1];
6          if d(a, p, c) < 0 then
7              break;
8          S pop;
9      S push p;
   // Prepare Second Pass
10 Stack[n] P' = S;
11 for j ← (n − 1) . . . 0 do
12     if P[j] ≠ S top then
13         P' push P[j];
14     else S pop;
   // Second Pass
15 for p ∈ P' do
16     while len(S) > 1 do
17         a ← S[len(S) − 2];
18         c ← S[len(S) − 1];
19         if d(a, p, c) < 0 then
20             break;
21         if d(a, p, c) = 0 and Δ_a c > Δ_a p then
22             break;
23         S pop;
24     S push p;
25 return S;
```

Angelehnt an Graham Scan Algorithmus. Gestrichen, weil zu schwer.

C.2 Stabiles Partitionieren (6 Punkte)

Beim Partitionieren eines Feldes A von Elementen nach einem gegebenen Prädikat P werden alle Elemente $e \in A$, welche die Eigenschaft $P(e)$ haben, an den Anfang der Liste verschoben. Das Ergebnisfeld A' ist also eine Permutation von A, für welches es einen maximalen Index k gibt, sodass folgendes gilt:

$$P(A'[i]) = \mathsf{true}, \text{ falls } i \leq k$$
$$P(A'[i]) = \mathsf{false}, \text{ falls } i > k.$$

Bei *stabilem* Partitionieren gilt zusätzlich, dass die relative Ordnung der Elemente innerhalb der Partitionen in A' gleich bleibt. Also für alle Elemente $e_1, e_2 \in A$ mit $P(e_1) = P(e_2)$ gilt, dass

$$\mathsf{index}(A', e_1) < \mathsf{index}(A', e_2) \quad \text{g.d.w.} \quad \mathsf{index}(A, e_1) < \mathsf{index}(A, e_2).^{7}$$

Implementieren Sie einen *in-place* Algorithmus (in Pseudocode), der in einem gegebenen Array A mit einem ebenfalls gegebenen Prädikat P eine stabile Partitionierung herstellt. Ihr Algorithmus darf die worst-case Laufzeitschranke von $\mathcal{O}(n^2)$ nicht überschreiten. Zeigen Sie, dass Ihr Algorithmus die Laufzeitbedingung erfüllt.

[7] $\mathsf{index}(A, e)$ steht hier für die Position des Elements e in Array A.

Zeigen Sie die Korrektheit Ihrer Implementierung mithilfe von Invarianten.

Algorithmus 3 : Stable Partition mit Bubblesort

```
1 for i ← [1 ... n] do
2     if P(a[i]) then
3         for j ← [i ... 1] do
4             if P(a[j − 1]) then
5                 assert P(a[0]) ∧ ··· ∧ P(a[j]);
6                 break;
7             swap(a[j], a[j − 1]);
8             invariant Σ_{k=0}^{j} P(a[k]) = Σ_{k=0}^{i} P(a[k]) = j + 1;
```

Äußere und innere Schleife jeweils maximal n Iterationen: $O(n^2)$.

Die Assertion in Zeile 5 gilt für $i = 0$. Gilt sie für einen Aufruf j, dann gibt es einen untersten zusammenhängenden Bereich $B = a[0] \dots a[j]$, für den das Prädikat gilt. Dieses B wurde beim j-ten Aufruf um eins nach oben vergrößert. Damit gilt die Assertion dann auch im nächsten Aufruf $j + 1$.

Die Invariante sagt aus, dass für jede Position i, jedes Element aus der Teilmenge $a[0]$ bis $a[i]$, das P erfüllt, im unteren Bereich B enthalten ist.

C.3 Directed Acyclic Graphs, DAG (7 Punkte)

Gegeben Sei ein gerichteter, *nicht notwendigerweise zusammenhängender* Graph $G = (V, E)$ mit n Knoten und m Kanten in Adjazenzfelddarstellung. Schreiben Sie einen Algorithmus isDAG (in Pseudocode), der genau dann true zurückgibt, wenn G ein DAG ist, und false zurückgibt, wenn G einen Zyklus enthält. Die Laufzeit darf $\mathcal{O}(n + m)$ nicht überschreiten, dies ist zu zeigen.

Algorithmus 4 : isDAG

```
1 ∀v ∈ V, visited[v] ← false;
2 ∀v ∈ V, visiting[v] ← false;
3 for v ∈ V do
4     if not isDAGInner(x) then
5         return false ;
6     return true;
```

Algorithmus 5 : isDAGInner

```
Input : Node v
1 if visited[v] then return true;
2 if visiting[v] then return false;
3 visiting[v] ← true;
4 for (v, x) ∈ E do
5     if not isDAGInner(x) then
6         return false ;
7 visiting[v] ← false;
8 visited[v] ← true;
9 return true;
```

Lösung über indegree und besuchte Knoten mitzählen/löschen auch gut. Breitensuche mit korrekter Level-Berechnung geht auch.