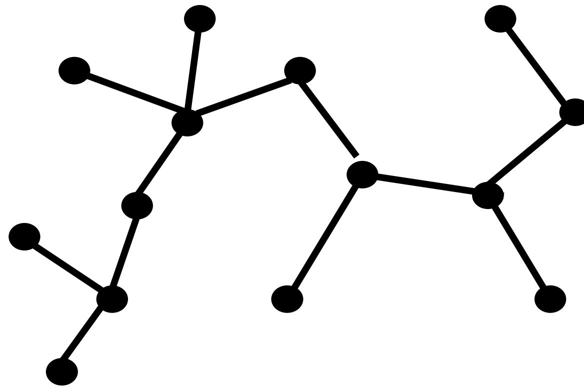
Grundlagen von Graphen

von Valentin Dallmeier



Begriffe und Definitionen..2
Der Abstrakte Datentyp Graph...5
Datenstrukturen für Graphen...6
Einfache Algorithmen auf Graphen..8
Literaturverzeichnis...10
Bildverzeichnis..10

1. Begriffe und Definitionen

1.1 Grundlegender Aufbau eines Graphen

Graphen werden verwendet um beliebige Objekte und deren Beziehungen zueinander darzustellen. So erfolgt beispielsweise in GPS-Systemen die Hinterlegung der zugrundeliegenden Straßenkarten in Form eines Graphen. In diesem Fall stellen Orte oder Kreuzungen die Objekte des Graphs und Verbindungen dieser Orte (z.B. in Form von Strassen) die Beziehungen dar.
Da eine große Anzahl von Problemen des täglichen Lebens mithilfe von Graphen repräsentiert werden können stellen sie eine der wichtigsten Datenstrukturen dar. Der Beginn der Graphentheorie wird im 18. Jahrhundert angesiedelt. Zur Lösung des berühmten Königsberger Brückenproblems stellte Euler im Jahre 1736 erstmals die Situation mithilfe eines Graphen dar und konnte so das Problem lösen.

Definition: Grundsätzlich wird ein Graph G durch ein paar zweier Mengen symbolisiert:

$$\mathbf{G = (V, E)}$$

- **V** ist die Menge aller Knoten (engl. *vertices*) die im Graphen enthalten sind. Zu beachten ist dass **V** nicht leer sein kann da ein Graph ohne Objekt wenig Sinn macht.
- **E** stellt die Menge aller Kanten (engl. *edges*) zwischen den Knoten des Graphs dar. Dabei wird eine Kante repräsentiert durch das Paar von Knoten die sie berührt. Graphen können auch nur Knoten und keine Kanten zwischen diesen haben. Eine Kante deren Beginn- und Endpunkt gleich sind heisst Schleife (engl. *loop*)

Beispiel:
Abbildung 1 zeigt ein Beispiel eines Graphen:

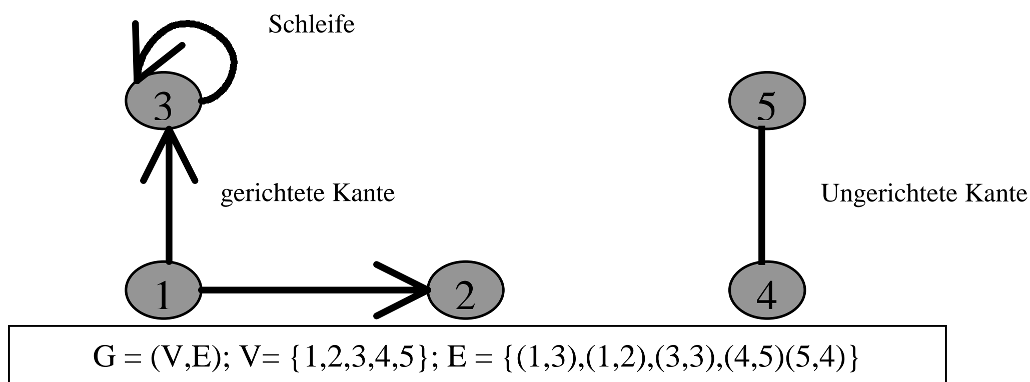


Abb. 1: Ein Beispiel-Graph

Innerhalb der Menge aller Kanten werden zwei Arten unterschieden:

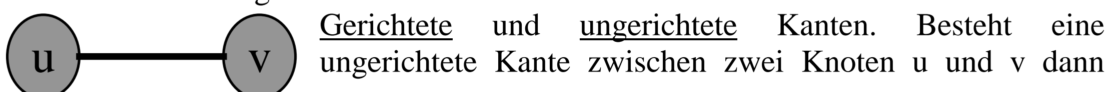


Abb. 2:ungerichtete Kante

Gerichtete und ungerichtete Kanten. Besteht eine ungerichtete Kante zwischen zwei Knoten u und v dann

bedeutet dies dass die Beziehung sowohl von u nach v als auch von v nach u gilt. Bei gerichteten Kanten gilt die Beziehung nur in eine Richtung. Zu beachten ist dass in einem Graphen gleichzeitig beide Arten von Kanten vorkommen können. Graphen die ausschließlich ungerichtete Kanten beinhalten heißen **ungerichtete Graphen**, wogegen Graphen deren Knoten nur durch gerichtete Kanten miteinander verbunden sind als **ungerichtete Graphen** bezeichnet werden.

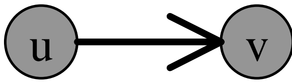


Abb. 3: gerichtete Kante

Für viele Probleme die mithilfe der Graphentheorie betrachtet werden ist es notwendig, jeder Kante einen Wert zuordnen zu können. Ein anschauliches Bespiel hierfür ist die Repräsentation eines Straßennetzes durch einen Graphen. Hier stellen die Knoten Kreuzungspunkte und die Kanten die Strassen zwischen den Kreuzungen dar. Das Gewicht einer Kante in solch einem System ist Entfernung zwischen den beiden Knoten die durch diese Kante verbunden werden.

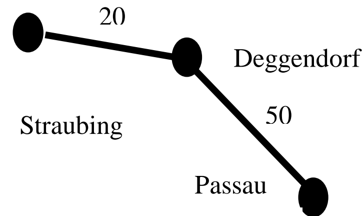


Abb. 4: Gewichteter Graph

1.2 Terminologie

Im Zuge der Entwicklung der Graphentheorie bildete sich eine eigene Terminologie für Sachverhalte rund um Graphen, Knoten und Kanten.

Definition: Adjazenz

Sind u, v zwei Elemente aus V und existiert (u,v) in E dann heißt v **adjazent** u. In einem Graphen bedeutet die Aussage „zwei Knoten sind adjazent" dass eine Kante zwischen diesen beiden Knoten existiert. Bei gerichteten Kanten gilt die Aussage nur in eine Richtung, bei ungerichteten Kanten hingegen ist die Relation symmetrisch.

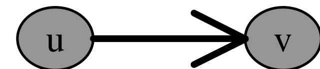


Abb. 5: Zwei adjazente Knoten

Definition: Grad eines Knoten

In einem ungerichteten Graphen ist der Grad eines Knoten die Anzahl der ihn berührenden Kanten. Dagegen wird in einem gerichteten Graphen zwischen ein- und ausgehenden Kanten unterschieden. Der out-degree eines Knoten ist die Zahl der den Knoten verlassenden Kanten und der in-degree entsprechend die Zahl der in den Knoten führenden Kanten.

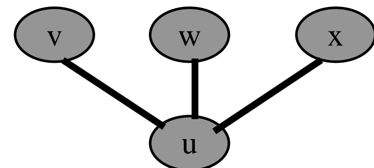


Abb. 6: Knoten u mit Grad(u) = 3

Definition: Pfad und zusammenhängender Graph

Eine Folge $<v_0 , ... , v_k>$ von Knoten heißt ein Pfad wenn (v_{i-1} , v_i) , i = 1,2...,k in der Menge der Kanten enthalten ist. Ein Pfad dessen Start- und Endpunkt übereinstimmen nennt man einen zyklischen Pfad. Ein Graph heißt zusammenhängend wenn für jedes Paar von Knoten innerhalb des Graphen mindestens ein Weg existiert. Existiert in einem Graph mindestens ein zyklischer Graph so wird er als zyklischer Graph bezeichnet.

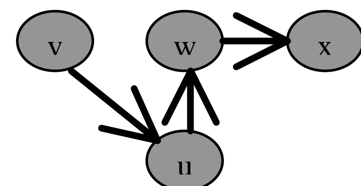


Abb.7: Pfad von v nach x <v.u.w.x>

Definition: „Subgraph“ und induzierter Subgraph
Ein Graph G’ = (V’, E’) wird als Subgraph eines Graphen G = (V, E) bezeichnet falls gilt: V’⊆ V und E’⊆ E. Eine Teilmenge V’’ eines Graphen induziert einen Subgraphen von G derart dass alle Kanten die kein Element aus V’’ berühren und alle Knoten die in V \ V’’ enthalten sind weggelassen werden.

1.3 Das Handshaking Lemma

Folgende Situation verdeutlicht eine Eigenschaft die jeder Graph besitzt:
Auf einer Party begrüßen sich alle Teilnehmer per Händedruck und jeder merkt sich wie oft er bzw. sie jemandem die Hand gegeben haben. Die Frage ist nun wie oft insgesamt an diesem Abend Hände geschüttelt wurden. Überträgt man dieses Problem auf die Graphentheorie so erhält man einen Graphen in dem jeder Knoten eine Kante zu jedem anderen aufweist. Nun lautet die Frage also wie groß die Summe über die Grade aller Knoten ist. Die Lösung dieses Problems bietet das nach dieser Situation benannte Handshaking-Lemma. Es besagt:

In einem Graphen ist die Summe über die Grade aller Knoten stets gleich der doppelten Zahl von Kanten im Graph.

Beweis:
Da diese Aussage für alle Graphen gelten muss bietet sich eine Induktion über die Zahl der im Graphen vorhandenen Knoten an:

IA: sei n=1
In einem Graphen mit nur einem Knoten existieren entweder 0 oder 1 Kante (Schleife). In beiden Fällen stimmt die Behauptung

IS: n -> n+1
Für den Subgraphen der durch die ersten n Knoten induziert wird gilt die Behauptung nach der Induktionsannahme. Sei k die Zahl der Kanten die den n+1. Knoten berühren. Jede dieser Kanten erhöht jeweils im Start- und Zielknoten den Grad um 1. Somit wird die Summe über alle Grade um 2k erhöht.

1.4 Spezielle Graphen und Subgraphen

Graphen die bestimmte Eigenschaften aus Abschnitt 1.2 erfüllen besitzen zum Teil eigene Namen:

Ein azyklischer, zusammenhängender ungerichteter Graph wird als **Baum** bezeichnet. Bäume sind somit in der Datenstruktur des Graphen mitenthalten und finden in der Informatik z.B. als binäre Suchbäume weit verbreitete Anwendungsmöglichkeiten.

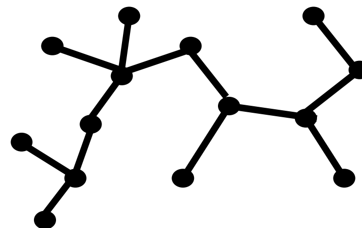


Abb. 8: Baum

Ein azyklischer ungerichteter Graph wird als **Wald** bezeichnet. Die Begriffsbildung resultiert aus der Tatsache dass die zusammenhängenden Subgraphen eines Waldes selbst wieder Bäume sind. Zu beachten ist ebenfalls dass ein Baum alleine schon einen Wald darstellt.

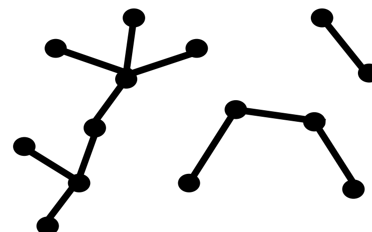


Abb. 9: Wald

2. Der abstrakte Datentyp (ADT) Graph

Ein abstrakter Datentyp ist ein Modell einer Datenstruktur das die Inhalte, Methoden und die Typen der Parameter festlegt. Dieses Modell dient also zur Spezifikation des Verhaltens der Methoden, nicht jedoch wie dieses Verhalten erreicht wird. Abbildung 10 enthält einen Auszug der Methoden dieses Datentyps:

Methode	Beschreibung
numVertices()	Gibt die Zahl der vorhandenen Knoten zurück
vertices()	Gibt einen Iterator über die im Graph vorhanden Knoten zurück
degree(v)	Gibt den Grad des Knoten v zurück
adjacentVertices(v)	Gibt einen Iterator über die zu v adjazenten Knoten zurück
incidentEdges(v)	Gibt einen Iterator über die v berührenden Knoten zurück
areAdjacent(v,w)	Prüft ob v und w adjazente Knoten sind
directedEdges()	Gibt einen Iterator über die gerichteten Kanten zurück
undirectedEdges()	Gibt einen Iterator über die ungerichteten Kanten zurück
origin(e)	Gibt den Startpunkt der Kante e zurück
inDegree(v)	Gibt den in-degree des Knoten v zurück
inIncidentEdges(v)	Gibt einen Iterator über alle in v endenden Knoten zurück
outAdjazentVertices(v)	Gibt einen Iterator über alle Knoten die Endpunkte von v verlassenden Kanten sind zurück
insertEdge(v,w,o)	Fügt eine Kante zwischen v und w ein und weißt ihr den Wert o zu.
insertVertex(o)	Fügt einen Knoten hinzu und speichert den Wert o in ihm
removeVertex(v)	Löscht den Knoten v aus dem Graphen
removeEdge(e)	Löscht die Kante e aus dem Graphen
makeUndirected(e)	Wandelt die Kante e in eine ungerichtete Kante um

Abb. 10: Tabellarische Übersicht über den ADT Graph

Für eine vollständige Darstellung verweise ich auf [GT] S. 544 ff.

3. Datenstrukturen für Graphen

Die Definition des abstrakten Datentypen bietet noch keine Informationen über die eigentliche Repräsentation innerhalb eines Computers. Für Graphen existieren mehrere Datenstrukturen die alle Vor- und Nachteile in sich bergen. Die wichtigsten 3 Ansätzen werden nun behandelt. Gemeinsam ist allen Strukturen dass die Menge der Knoten jeweils als Liste gespeichert wird. Abbildung 11 zeigt ein Vertex-Objekt:

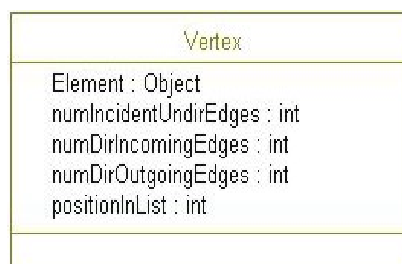


Abb. 11: Vertex-Objekt

3.1 Kantenliste

Die Kantenliste stellt die einfachste Datenstruktur da. Wesentliches Merkmal ist hierbei dass jede Kante explizit als Objekt gespeichert wird:

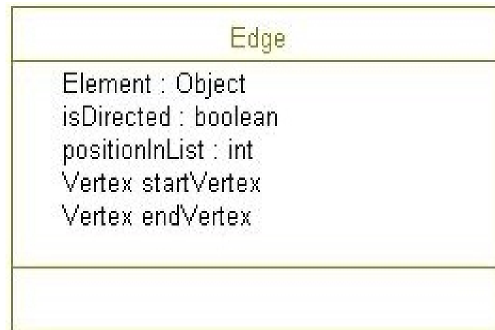


Abb. 12: Edge-Objekt

Jede Kante kennt also den Knoten an dem sie beginnt und den Endknoten. Die Menge aller Kanten innerhalb des Graphen wird typischerweise ebenfalls als Liste hinterlegt. Durch diese Realisierung wird der direkte Zugriff von einer Kante auf die Knoten die sie berührt ermöglicht. Abbildung 12 zeigt einen Beispielgraphen und eine Veranschaulichung der Repräsentation mithilfe der Kantenlisten-Struktur

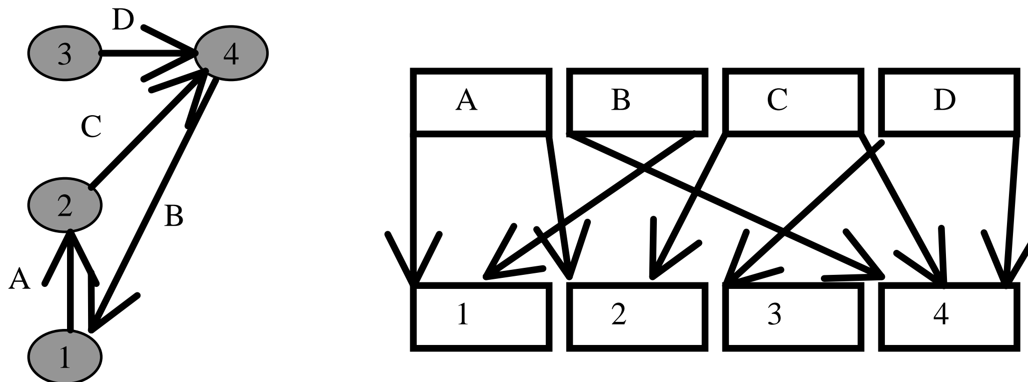


Abb. 12: Veranschaulichung der Kantenlisten-Struktur

Ein Nachteil dieser Darstellung ist dass der direkte Zugriff von einem Knoten aus auf die ihn berührenden Kanten nicht möglich ist. Dies erfordert hier einen kompletten Durchlauf der Kantenliste. Somit ist die Kantenliste für Methoden die den Zugriff Knoten-Kante brauchen nur bedingt geeignet. Ist G ein Graph mit **m** Kanten und **n** Knoten so ergeben sich folgende Komplexitäten:

Operation	Laufzeit
numVertices, numEdges	O(1)
areAdjacent, adjacentVertices, incidentEdges, inIncidentEdges	O(m)
InsertVertex, insertEdge, makeUndirected, removeEdge	O(1)
removeVertex	O(m)

Abb 13: Laufzeitkomplexitäten bei Verwendung der Kantenlisten-Struktur

3.2 Adjazenzliste

Die Adjazenzlistenstruktur stellt eine Erweiterung der Kantenliste dar. Dabei werden zusätzlich an jedem Knoten die ihn berührenden Kanten abgespeichert. Somit ist dann auch der Zugriff Knoten-Kante möglich was die Laufzeit-Komplexität einiger Methoden des ADT wesentlich beschleunigt. Die Adjanzenzlisten werden oft als doppelt verkettete Listen realisiert. Somit muss beispielsweise zur Bestimmung der Nachbarn von v (= zu v adjazente Knoten) nur noch durch die Adjazenzlisten von v durchgelaufen werden.

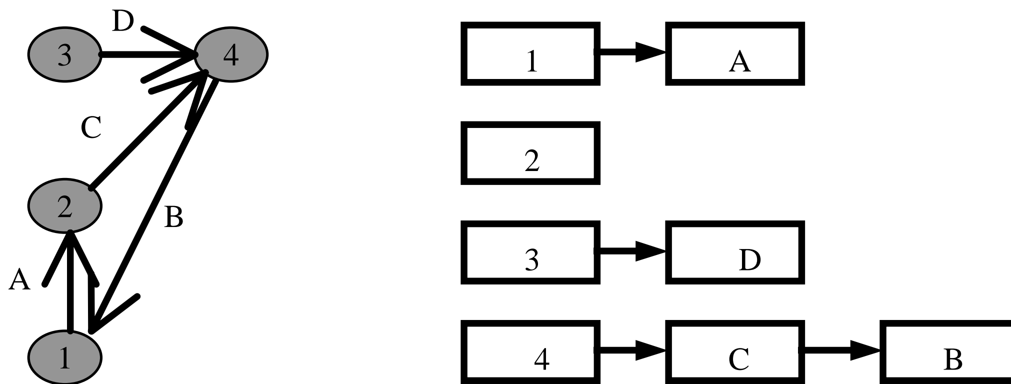


Abb. 13: Veranschaulichung der Adjazenzlisten-Struktur

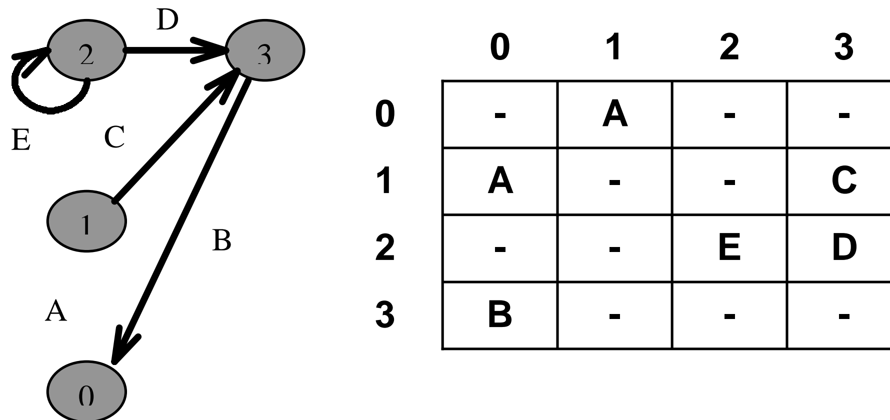
Operation	Laufzeit
numVertices, numEdges	O(1)
adjacentVertices, incidentEdges, inIncidentEdges	**O(deg(v))**
areAdjacent(v,w)	**O(min(deg(u),deg(v))**
insertVertex, insertEdge, makeUndirected, removeEdge	O(1)
removeVertex	O(m)

Abb. 14: Laufzeiten bei Verwendung der Kantenliste

Der Spezialfall eines Knoten der mit allen anderen Knoten im Graph verbunden ist führt wieder zu einer Laufzeit proportional zur Zahl der Kanten.

3.3 Adjazenzmatrix

Der Grundgedanke hinter dieser Datenstrukur ist die Verwendung eines zwei-dimensionalen Arrays zur Speicherung der Kanten. Dieses Array ist in beiden Dimensionen über die im Graph gespeicherten Knoten indiziert. Existiert zwischen den beiden Knoten u und v eine Kante so wird in der Adjazenzmatrix an der Stelle [u,v] ein Eintrag mit dieser Kante gespeichert. Um zu prüfen ob zwei Knoten adjazent sind muss somit nur noch auf das Array an der entsprechenden Position zugegriffen werden. Ein Nachteil dieser Art der Repräsentation istder gestiegene Speicherbedarf. Durch die Speicherung in einem zwei-dimensionalen Array ergibt sich eine quadratische Speicherkomplexität.



	0	1	2	3
0	-	A	-	-
1	A	-	-	C
2	-	-	E	D
3	B	-	-	-

Abb. 15: Veranschaulichung der Adjazenzmatrix

Die Wahl der Datenstruktur wird im Wesentlichen durch die für den Graphen vorgesehene Anwendung bestimmt. Typischerweise verwendet die Tiefensuche auf einem Graphen besonders häufig die Methode adjacentVertices() um die Nachbarn eines Knoten zu bestimmen. Hierfür bieten sich sowohl die Adjazenzlisten- als auch die Adjazenzmatrix-Struktur an da bei beiden Verfahren diese Methode eine Laufzeit proportional zum Grad des jeweiligen Knoten hat wohingegen diese Methode bei der Kantenliste langsamer abläuft.

4. Einfache Algorithmen auf Graphen

4.1 Topologische Sortierung

Die topologische Sortierung wird auf gerichtete azyklische Graphen angewendet. Jeder Graph dieser Art definiert durch seine Struktur eine Halbordnung auf den ihn ihm gespeicherten Knoten. Eine topologische Sortierung eines Graphen ist eine lineare Anordnung der Knoten welche die definierte Halbordnung respektiert. Von Bedeutung ist an dieser Stelle dass diese Anordnung nicht eindeutig ist, d.h. es gibt mehrere Sortierungen die alle die Halbordnung nicht verletzen.

Pseudocode:

- *Der in-degree aller Knoten wird ermittelt* **(O(n))**
- *Die Knoten mit dem in-degree 0 werden in die Liste eingefügt*
- *Solange die Liste nicht leer ist:*
    - *Der erste Knoten wird aus der Liste entnommen und der Sortierung hinzugefügt*
    - *Der in-degree aller benachbarten Knoten wird um 1 erniedrigt* **(O(deg(v))**
    - *Ist der in-degree eines Nachbarknotens jetzt 0 so wird er in die Liste eingefügt*

Die Betrachtung der Laufzeitkomplexität des vorgestellten Pseudocodes ergibt folgendes: Für jeden Knoten innerhalb des Graphen muss einmal durch die zu ihm adjazenten Knoten gelaufen werden. Da die topologische Sortierung nur auf gerichteten Graphen arbeitet gibt es durch jede Kante jeweils eine Verbindung von einem Knoten zu einem weiteren. Somit ergibt sich **O(n*m)** als Laufzeit für den Algorithmus.

4.2 Zyklenfreiheit

Für die oben aufgeführte topologische Sortierung ist es von Bedeutung dass der zu verarbeitende Graph keine Zyklen (vgl. 1.2) enthält da sonst der Algorithmus nicht anwendbar ist da keine Halbordnung definiert wird. Das nachstehende Verfahren prüft einen Graphen dahingehend.

Die Klasse Knoten wird um ein zusätzliches Feld erweitert dass die Unterscheidung von 3 Zuständen ermöglicht:

- Der Knoten ist noch unbesucht
- Der Knoten ist momentan in Bearbeitung
- Der Knoten ist bereits abgearbeitet

Der erste Teil des Algorithmus prüft mit Hilfe von Rekursion ob von einem Knoten ausgehend ein Zyklus im Graphen existiert:

Pseudocode 1:

HasCycle(v):

- *Der Knoten v wird als in Bearbeitung markiert*
- *Für alle zu v adjazenten Knoten **w** wird überprüft*
    - *Ist der Knoten schon als in Bearbeitung markiert -> return **true!***
    - *Wurde der Knoten noch nicht besucht -> return hasCycle(**w**)*
- *Return **false***

Mit dieser Prüfung ist der Test allerdings noch nicht abgeschlossen da durchaus noch zyklische Pfade von anderen Knoten innerhalb des Graphen existieren können.

Pseucdocode 2:

HasCycle(G):

- Alle Knoten innerhalb von G werden als noch nicht besucht markiert
- Für jeden Knoten v
    - Wenn v noch nicht besucht wurde -> return hasCycle(**v**)
- Return **false**

Dadurch werden alle Komponenten des Graphen auf Zyklenfreiheit überprüft.

5. Literaturverzeichnis

[GT] M. T. Goodrich, R. Tamassia:
Data Structures and Algorithms in Java
John Wiley & Sons Inc, 1998
(Sehr anschauliche Einführung mit vielen Bildern, Java-spezifisch)

[RD] Reinhard Diestel
Graph Theory (Electronic Edition)
Springer Verlag, 2000
(Graphen-Theorie eher von der mathematischen Seite, frei verfügbar unter
http://www.math.uni-hamburg.de/home/diestel/books/graph.theory)

[AHU] A. Aho; J.E. Hopcroft und J.D. Ullman
Data Structures and Algorithms
Addison-Wesley Publishing Company, Reading Massachusetts, 1983
(Ebenfalls sehr gut zu lesen. In der Bibliothek allerdings lieber die neuere Ausgabe aus dem Semesterapparat nehmen)

6. Bildquellen

Alle in diesem Dokument vorkommenden Abbildungen wurden vom Autor selbst erstellt.