

Teil 5: Algorithmen auf Graphen

Graphtraverse

Graphtraverse

In vielen Anwendungen ist es notwendig einen gegebenen Graphen systematisch zu durchsuchen, bzw. für jeden Knoten und/oder jede Kante eine bestimmte Berechnung durchzuführen.

Wenn alle Knoten und Kanten des Graphen bekannt sind, kann dies teilweise durch Ablaufen der jeweiligen Listen geschehen.

In vielen Anwendungen ist jedoch ein anderes Herangehen effizienter oder notwendig.

Weiß man z.B., dass der gesuchte Knoten in der Nähe eines bereits bekannten Knotens ist, ist es sinnvoll die Nachbarschaft systematisch zu durchsuchen.

In anderen Fällen ist der Graph nicht vollständig bekannt, bzw. passt nicht vollständig in den Speicher (z.B. Web-Crawler).

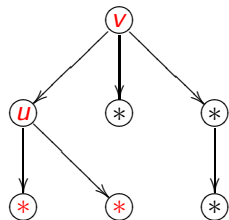
Für solche Fälle eignen sich die **Tiefen- und Breitensuche** oder Modifikationen.

Teil 5: Algorithmen auf Graphen
Graphtraverse

Tiefensuche

Die Tiefensuche

Bei der **Tiefensuche (Depth-First-Search, DFS)** gehen wir von einem Knoten zuerst in die Tiefe.



Wenn wir einen Knoten v besuchen, wählen wir einen seiner noch unbesuchten Nachfolger u aus und führen im nächsten Schritt eine Tiefensuche, beginnend bei u aus.

Damit besuchen wir die Nachfolger von u bevor seine „Geschwister“, d.h. die anderen Kinder von v besucht werden.

Dies kann entweder mittels einer Rekursion oder eines Stacks realisiert werden.

Prozedur (DFS mittels Rekursion)

Eingabe: *Ein Graph $G = (V, E)$*

Daten: *Ein Feld $\text{scanned}[1 \dots |V|]$ von Markierungen.*

für $v = 1, \dots, n$ **tue** $\text{scanned}[v] := 0$

solange ein Knoten v mit $\text{scanned}[v] = 0$ existiert **tue**

$\text{DFS_visit}(v, G, \text{scanned})$

Ende

Prozedur (DFS_visit)

Eingabe: *Graph $G = (V, E)$, $v \in V$ und $\text{scanned}[1 \dots |V|]$*

$\text{scanned}[v] := 1$

für alle $u \in \text{succ}[v]$ mit $\text{scanned}[u] = 0$ **tue**

$\text{DFS_visit}(G, u, \text{scanned})$

Ende

Laufzeit der Tiefensuche

Satz

Die Prozedur DFS benötigt zur Durchmusterung eines Graphen $G = (V, E)$ $O(|V| + |E|)$ Schritte.

Beweis

Die Initialisierung kostet $O(|V|)$, denn für jeden Knoten muss `scanned[v]` auf 0 gesetzt werden.

Beginnen wir die Tiefensuche mit Knoten 1, können wir bei der Rückkehr in die Prozedur einfach den nächsten unbesuchten Knoten in numerischer Reihenfolge suchen. Damit muss bei der Suche nach dem nächsten unerforschten Knoten jeder Knoten nur einmal inspiziert werden. Dies führt zu einer Gesamtlaufzeit von $O(|V|)$.

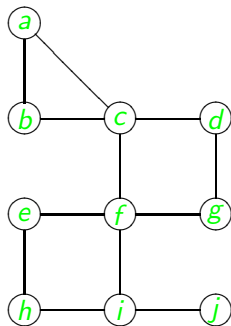
Beweis (Laufzeit der Tiefensuche - Fortsetzung)

Der Zeitbedarf für einen Aufruf von DFS_visit, ohne die Zeit für die rekursiven Aufrufe, ist durch proportional zu der Anzahl der Nachfolger des besuchten Knotens.

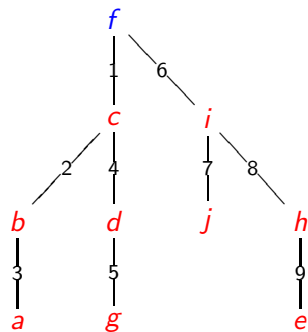
Da DFS_visit für jeden Knoten genau einmal aufgerufen wird, ergibt sich somit ein Zeitbedarf von $O(|V| + |E|)$.

Insgesamt ergibt sich für DFS somit ein Zeitaufwand von $O(|V| + |E|)$. \square

Tiefensuche - Ein Beispiel



Startknoten: *f*



Teil 5: Algorithmen auf Graphen Graphtraverse

Breitensuche

Die Breitensuche

Während bei der Tiefensuche erst die Nachfolger eines Nachfolgers besucht wurden, werden bei der **Breitensuche (Breadth-First-Search, BFS)** zuerst alle Nachfolger besucht, bevor deren Nachfolger bearbeitet werden.

Die Breitensuche können wir ohne Rekursion mittels einer Warteschlange (Queue) implementieren.

Prozedur (BFS)

Eingabe: *Ein Graph $G = (V, E)$*

Daten: *Ein Feld $\text{scanned}[1 \dots |V|]$ von Markierungen und eine Queue Q .*

für $v = 1, \dots, n$ **tue** $\text{scanned}[v] := 0$

solange *ein Knoten v mit $\text{scanned}[v] = 0$ existiert* **tue**
 $\text{scanned}[v] := 1, Q := (Q, v)$

solange $Q \neq \emptyset$ **tue**

Entnehme das erste Element v aus Q

für alle $u \in \text{succ}[v]$ *mit* $\text{scanned}[u] = 0$ **tue**

// Besuche alle unbesuchten Nachfolger

$\text{scanned}[u] := 1, Q := (Q, u)$

Ende

Ende

Ende

Laufzeit der Breitensuche

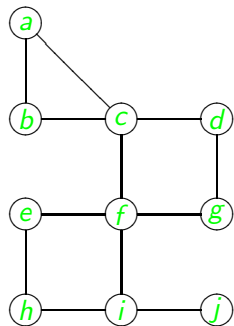
Wie bei der Tiefensuche ergibt sich

Satz

Die Prozedur BFS benötigt zur Durchmusterung eines Graphen $G = (V, E)$ $O(|V| + |E|)$ Schritte.

Analog zur Breitensuche mit einer Queue kann man die Tiefensuche ohne Rekursion mit einem **Stack** realisieren.

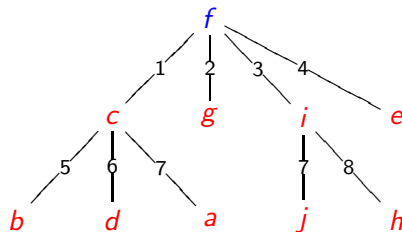
Breitensuche - Ein Beispiel



Startknoten: *f*

Queue:

c,g,i, eg, i, eg, i, e, *b,d*, ai, e, b, d, ae, b, d, ae, b, d,



Teil 5: Algorithmen auf Graphen

Topologische Ordnung

Topologische Ordnung

In vielen Anwendungen sind gerichtete azyklische Graphen von Bedeutung. Dabei spielt insbesondere die Existenz einer **topologischen Ordnung** auf einem solchen Graphen eine große Rolle.

Definition (Topologische Ordnung)

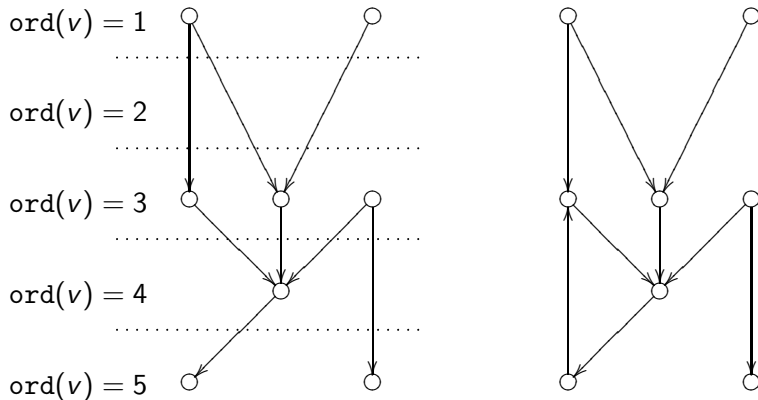
Eine **topologische Schichtung** auf einem gerichteten Graphen $G = (V, E)$ ist eine Abbildung $\text{ord}: V \rightarrow \{1, \dots, n\}$ mit $n = |V|$, so dass für jede Kante $(u, v) \in E$ die Ungleichung $\text{ord}(u) < \text{ord}(v)$ gilt.

Ist die Abbildung ord bijektiv, spricht man von einer **topologischen Ordnung**.

Eine **topologische Schichtung** kann somit als eine Einteilung der Knoten in maximal n **Schichten** interpretiert werden, so dass eine Kante nur von einer niedrigeren in eine höhere Schicht führt.

Eine **topologische Ordnung** ist eine spezielle Schichtung, bei der jede Schicht genau einen Knoten enthält.

Topologische Schichtung - ein Beispiel



Topologische Schichtungen und Ordnungen

Offensichtlich ist jede topologische Ordnung eine Schichtung. Aber wie ist es umgekehrt?

Aus einer gegebenen Schichtung ord kann leicht eine topologische Ordnung konstruiert werden, in dem die Knoten innerhalb einer Schicht beliebig geordnet werden.

Sei n_i die Anzahl der Knoten in der i -ten Schicht, d.h. der Knoten mit $\text{ord}(v) = i$. Dann hat jeder Knoten v zwei Ordnungszahlen, einmal seine Schicht $\text{ord}(v)$ und zum zweiten eine Nummer $f(v)$ mit $1 \leq f(v) \leq n_{\text{ord}(v)}$ innerhalb seiner Schicht.

Die Gesamtordnung ord' ergibt sich dann als:

$$\text{ord}'(v) := \sum_{j=1}^{\text{ord}(v)-1} n_j + f(v).$$

Anders ausgedrückt: Die Schichten werden einzeln geordnet, und dann ihre Ordnungen einfach „aneinander gehängt“.

Existenz von topologischen Ordnungen

Lemma (Existenz von topologischen Ordnungen)

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Ordnung, wenn er azyklisch ist.

Beweis

Nehmen wir zuerst an, dass G eine topologische Ordnung ord besitzt und einen Kreis $(u, v_1, \dots, v_{l-1}, u)$ enthält. Dann gilt offensichtlich:

$$\text{ord}(u) < \text{ord}(v_1) < \dots < \text{ord}(v_{l-1}) < \text{ord}(u)$$

was offensichtlich ein Widerspruch ist!

Beweis (Existenz von topologischen Ordnung (Fortsetzung))

Jetzt wollen wir zeigen, dass wir eine topologische Ordnung konstruieren können, wenn G azyklisch ist. Dazu führen wir eine Induktion über die Anzahl der Knoten durch.

Falls $n = 1$, ist die Existenz einer topologischen Ordnung offensichtlich.

Sei nun $n > 1$. Zuerst behaupten wir, dass ein Knoten u in G existiert mit $d_{in}(u) = 0$. Dies beweisen wir indirekt.

Angenommen es gibt keinen solchen Knoten. Dann können wir zu jedem Knoten einen Vorgänger finden. D.h. wenn wir bei einem beliebigen Knoten beginnen, können wir stets einer Kante rückwärts zu einem anderen Knoten folgen.

Spätestens nachdem wir n verschiedene Knoten besucht haben (alle die es gibt), müssen wir zu einem Knoten zurück kehren, den wir bereits besucht hatten. Damit enthält G aber einen Kreis, was im Widerspruch zu seiner Azyklizität steht.

Beweis (Existenz von topologischen Ordnung (Fortsetzung))

Damit müssen wir irgendwann auf einen Knoten u stoßen, der keinen Nachfolger besitzt, d.h. $d_{in}(u) = 0$.

Sei nun $G' = G - u$. Offensichtlich ist G' azyklisch und somit existiert nach Induktionsvoraussetzung eine topologische Ordnung $\text{ord}' : V \setminus \{u\} \rightarrow \{1, \dots, n-1\}$ von G' .

Auf G definieren wir $\text{ord} : V \rightarrow \{1, \dots, n\}$ als

$$\text{ord}(v) := \begin{cases} \text{ord}'(v) + 1 & \text{falls } v \neq u \\ 1 & \text{falls } v = u. \end{cases}$$

Diese Abbildung ist eine topologische Ordnung, denn für alle Kanten (v, v') mit $v, v' \neq u$ gilt $\text{ord}(v) = \text{ord}'(v) + 1 < \text{ord}'(v') + 1 = \text{ord}(v')$ und für alle Kanten (u, v) gilt $\text{ord}(u) = 1 < 2 \leq \text{ord}'(v) + 1 = \text{ord}(v)$. Kanten der Form (v, u) gibt es nicht, da $d_{in}(u) = 0$. □

Die Existenz einer topologischen Ordnung

Der vorangegangene Beweis ist bereits ein Algorithmus zur Konstruktion einer topologischen Ordnung.

Allerdings würde die Suche eines Knotens u mit $d_{in}(u) = 0$ nach dem obigen Verfahren im schlechtesten Fall $O(n)$ Zeit kosten. Dadurch ergibt sich eine Gesamtlaufzeit von $O(n^2)$.

Stattdessen werden wir stets eine Liste aller Knoten mit **verbliebenen** Eingangsgrad 0 mitführen.

Dazu werden wir jedesmal, wenn ein Knoten ohne Vorgänger entnommen wird, den Grad seiner Nachfolger entsprechend herabsetzen.

Wie man leicht sieht führen wir im Prinzip eine modifizierte Breitensuche durch, bei der wir die Knoten in der Reihenfolge nummerieren, wie wir sie besuchen.

Algorithmus (Topologische Ordnung)

Eingabe: Ein gerichteter Graph $G = (V, E)$.

Ausgabe: Eine topologische Ordnung ord auf G , falls G azyklisch ist.

Daten: Integer-Felder $in[1 \dots n]$ und $ord[1 \dots n]$ und eine Knoten-Liste L .

für $v := 1 \dots n$ **tue**

$in[v] := d_{in}[v]$

wenn $in[v] = 0$ **dann** $L := (L, v)$

Ende

$o := 0$

solange $L \neq \emptyset$ **tue**

 Entnehme ein v aus L

$o := o + 1, ord[v] := o$

für $e = (v, u) \in out(v)$ **tue**

$in[u] := in[u] - 1$

wenn $in[u] = 0$ **dann** $L := (L, u)$

Ende

Ende

wenn $o = n$ **dann** gebe aus: „ G ist azyklisch“

sonst gebe aus: „ G ist nicht azyklisch“

Die Laufzeit

Verwendet man Adjazenzlisten für die Darstellung von G , ergeben sich die Laufzeiten für die einzelnen Phasen folgendermaßen:

- 1 Die Initialisierung benötigt $O(|V| + |E|)$, um die Eingangsgrade zu bestimmen und die Liste zu füllen.
- 2 Die Schleife wird höchstens $|V|$ -mal durchlaufen, da ein Knoten erst behandelt wird, wenn alle seine Vorgänger bereits bearbeitet worden sind. Dadurch kann er später nicht wieder in die Liste L eingefügt werden.
- 3 Innerhalb der schleife wird jede Kante höchstens einmal betrachtet. dies geschieht genau dann, wenn ihr Startknoten gerade bearbeitet wird.
- 4 Damit ergibt sich die Laufzeit für die Schleife als $O(|V| + |E|)$.
- 5 der Abschließende Test benötigt $O(1)$.

Die Laufzeit ist somit klar. Aber wie sieht es mit der Korrektheit aus? Zuerst beobachten wir, dass ein Knoten v , der auf einem Kreis $(v, v_1, \dots, v_{l-1}, v)$ liegt, nie zur Liste L hinzugefügt wird. Damit dies geschehen würde, müssten sämtliche seiner Vorgänger bereits besucht worden sein, d.h. auch v_{l-1} . Dafür müsste aber wiederum v_{l-2} usw. bis v_1 und schließlich v selbst bereits besucht sein.

Damit besucht der Algorithmus nur Knoten, die auf keinem Kreis liegen. Für diese wird außerdem offensichtlich eine topologische Ordnung konstruiert.

Satz

Der Algorithmus testet in Zeit $O(|V| + |E|)$, ob ein gerichteter Graph azyklisch ist. Gleichzeitig konstruiert er eine topologischen Ordnung, wenn möglich.