Karlsruher Institut für Technologie — Prof. Dr. Jörn Müller-Quade
Institut für Theoretische Informatik — Björn Kaidel, Sebastian Schlag, Sascha Witt

9. Tutorenblatt zu Algorithmen I im SoSe 2017

http://crypto.iti.kit.edu/index.php?id=799
{bjoern.kaidel,sascha.witt}@kit.edu

In der Vorlesung wurden Graphen, Graphrepräsentationen, Breitensuche und Tiefensuche behandelt. Die grundlegenden Begriffe sollten wiederholt werden. Einführende Aufgaben findet ihr z.B. in Cormen et al. - Introduction to Algorithms.

Aufgabe 1 (*Graphrepräsentation*)

Gegeben ein Beispielgraph. Gib jeweils die Repräsentation mit Hilfe von Kantenfolgen, Adjazenzfeldern, Adjazenzlisten und Adjazenzmatrix an. Welche Vor- und Nachteile haben die verschiedenen Repräsentationen? (insbesondere in Hinsicht auf Laufzeitverhalten, Speicherplatzverbrauch etc.)

Aufgabe 2 (*Graphtraversierung*)

Gegeben ein Beispielgraph. Führe eine Breitensuche und eine Tiefensuche auf dem gegebenen Graph durch. Klassifiziere jeweils die Kanten (tree, backward, cross, forward).

Aufgabe 3 (*Gerichteter azyklischer Graph - „Graphenbeweise“*)

Es sei $G = (V, E)$ ein gerichteter azyklischer Graph (DAG) mit endlich vielen und mindestens einem Knoten. Zeige, dass G mindestens einen Knoten mit Eingangsgrad 0 besitzt.
Anmerkung: Evtl. ist es sinnvoll, noch ein paar mehr solche Aussagen zu beweisen, damit sich die Studierenden an sowas gewöhnen.

Musterlösung:
Angenommen jeder Knoten hat einen Eingangsgrad ≥ 1. Sei $v_0 \in V$ ein beliebiger Knoten. Da v_0 Eingangsgrad $\neq 0$ hat, existiert mindestens eine Kante $(v_1, v_0) \in E$ mit $v_1 \in V$. Wir haben nun einen Pfad $v_1 \to v_0$ im Graph gefunden. Diesen Pfad können wir beliebig verlängern, da auch v_1 wieder Eingangsgrad $\neq 0$ hat, der darauffolgende Knoten auch etc. Somit können wir für beliebige $n \in \mathbb{N}$ einen Pfad

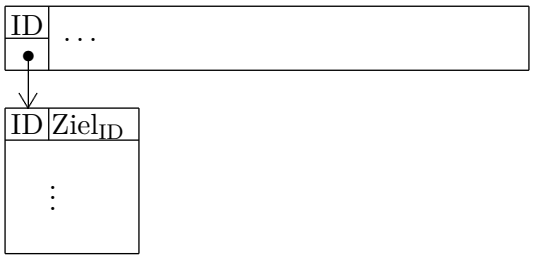
$$v_n \to v_{n-1} \to ... \to v_1 \to v_0$$

innerhalb des Graphen finden. Da $|V|$ endlich ist, müssen für alle $n > |V|$ zwei Indizes $i \neq j$ von Knoten auf dem Pfad existieren, sodass $v_i = v_j$ gilt, d.h. der Pfad muss einen Zyklus enthalten. Dies ist ein Widerspruch, da G azyklisch ist und somit muss mindestens ein Knoten mit Eingangsgrad 0 existieren.

Aufgabe 4 (*Kreativaufgabe*)

Gegeben sei ein gerichteter, stark zusammenhängender Graph in folgender Darstellung:

- Ein Knotenarray, das zu jedem Knoten v einen Eintrag mit seiner ID und einen Zeiger auf ein Array mit den von v ausgehenden Kanten enthält. Die Knoten haben eindeutige IDs.
- Das Kantenarray mit den ausgehenden Kanten von v enthält für jede Kante e einen Eintrag mit ihrer ID und der ID des Zielknotens der Kante.



Auf diesem Graphen soll nun eine BFS ausgeführt werden, wobei zu jedem Zeitpunkt nur $\mathcal{O}(1)$ zusätzlicher Speicher verwendet werden soll. Die Laufzeit darf dabei schlechter als bei der üblichen BFS sein. Die Art der Darstellung des Graphen soll während der BFS erhalten bleiben, es ist jedoch erlaubt z.B. die Knoten im Knotenarray zu permutieren.
Geben Sie eine Pseudocode-Implementierung der BFS an, die diese Bedingungen erfüllt und begründen Sie, warum diese Implementierung nur $\mathcal{O}(1)$ zusätzlichen Speicher benötigt. Es soll dabei für jeden Knoten eine unbekannte Funktion f aufgerufen werden, die als Eingabe die Knoten-ID und die Ebene (Entfernung zum Startknoten) des Knotens hat. Es kann angenommen werden, dass f während der Ausführung $\mathcal{O}(1)$ und nach der Ausführung keinen Speicher benötigt.

Musterlösung:

```
1:  procedure BFS(NodeArray[1...n] nodes, NodeID start)
2:      Finde i mit nodes[i].ID = start
3:      Vertausche nodes[i] mit nodes[1]
4:      Setze q1 := 1, q2 := 2, ebene := 0 und u := 2
5:      while q1 ≤ n do
6:          while q1 < q2 do
7:              call f(nodes[q1].ID, ebene)
8:              forall Kanten k im Kantenarray von nodes[q1] do
9:                  Suche i ∈ {u, ..., n} mit nodes[i].ID = Ziel_ID
10:                 If i gefunden do
11:                     Vertausche nodes[i] mit nodes[u]
12:                     u++
13:             q1++
14:         q2 := u, ebene++
15: return
```

Die Implementierung benötigt nur zusätzlichen Platz für die Indizies q_1, q_2, i und u und für die Ebenennummer und den Index im Kantenarray des gerade bearbeiteten Knotens. Der Platzbedarf für diese 6 Zahlen ist nach Definition konstant. Zudem wird noch ein Speicherplatz für einen Knoten zum Vertauschen benötigt.
Die Implementierung bringt alle Knoten immer im Array nodes unter, ohne dessen Größe zu ändern. Im Bereich $1, \ldots, q_1 - 1$ liegen die schon bearbeiteten Knoten, im Bereich $q_1, \ldots, q_2 - 1$ liegt die Queue der in dieser Ebene noch zu bearbeitenden Knoten, im Bereich $q_2, \ldots, u - 1$ liegt die Queue der in der nächsten Ebene zu bearbeitenden Knoten und im Bereich $u, \ldots, n$ liegen die noch nicht gefundenen Knoten.