Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2017
Brock / Mabrouk / Sieverling / Stahl

Aufgabenblatt 5

letzte Aktualisierung: 29. May, 10:17 Uhr

Ausgabe: 29.05.2016
Abgabe: 07.06.2016 19:59

Thema: *Graphen*

Abgabe

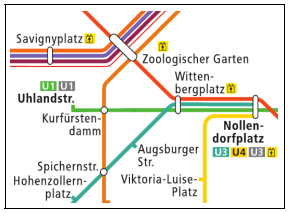
Die folgenden Dateien müssen für eine erfolgreiche Abgabe im svn Ordner `Tutorien/txx/Studierende/deinname@TU-BERLIN.DE/Abgaben/` eingecheckt sein:

Geforderte Dateien:	
`Blatt05/src/DiGraph.java`	Aufgabe 2.1-2.5

Als Abgabe wird nur die neueste Version im svn gewertet.

1. Aufgabe: Vergleich von Graphenalgorithmen

Gegeben ist folgender Ausschnitt aus dem Fahrplan der Berliner U-Bahn:



U2	↓	↑
Zoologischer Garten	11:15	11:12
Wittenbergplatz	11:17	11:10

U9	↓	↑
Zoologischer Garten	11:25	11:38
Kurfürstendamm	11:26	11:37
Spichernstr.	11:27	11:36

U1	↓	↑
Uhlandstr.	11:22	11:37
Kurfürstendamm	11:24	11:35
Wittenbergplatz	11:28	11:31

U3	↓	↑
Wittenbergplatz	10:48	11:09
Augsburger Str.	10:50	11:07
Spichernstr.	10:54	11:03

1.1. (Übung) Darstellung als Graph Stellt die Fahrpläne als ungerichteten, gewichteten Graphen dar. Verwendet die Fahrzeit zwischen zwei Stationen als Kantengewicht. Es wird angenommen, dass die Fahrzeit in beide Richtungen gleich ist und sich im Laufe des Tages nicht ändert. Die Zeiten zum Umsteigen und Wartezeiten werden vernachlässigt.

1.2. (Tut) Tiefensuche - Breitensuche Wie funktionieren Tiefen- und Breitensuche. Was sind die Unterschiede dazwischen? Was sind die Anwendungsfälle?

1.3. (Tut) Tiefensuche - Prinzip Simuliert die Breitensuche für den vorgegebenen Graph.

Schritt	akt. Knoten	Queue
0		Au
1		
2		
3		
4		
5		
6		

1.4. (Tut) Tiefensuche - Prinzip Tiefensuche ist ein Graphalgorithmus, der ähnlich zur Breitensuche, allerdings mit einem Stack, funktioniert. Das bedeutet wir erkunden einen Pfad erst so weit wir können, bevor wir die nächste Abzweigung betrachten.

Schritt	akt. Knoten	Stack
0		Au
1		
2		
3		
4		
5		
6		

1.5. Tiefensuche Implementieren (50 Punkte) Implementiert nun den Tiefensuch-Algorithmus im vorgegebenen Methodenrumpf `DiGraph.depthFirstSearch(Node startNode)`. Die Methode soll eine Liste mit Knoten zurückgeben, die so geordnet sind, wie euer Algorithmus diese Knoten besucht hat.

Hinweis: Es kann sein, dass die Tiefensuche nicht alle Knoten erreicht, falls der Graph nicht stark zusammenhängend ist. Euer Algorithmus soll dabei nur alle vom Startknoten aus erreichbaren Knoten bearbeiten, alle unerreichbaren Knoten können ignoriert werden.
Beim Implementieren der Hausaufgaben könnt ihr die Visualisierungsmethoden benutzen um die Ausführung des Algorithmus und seine Arbeit schrittweise sichtbar zu machen:

- `stopExecutionUntilSignal` hier wartet euer Algorithmus auf einen Klick auf die Graph-Visualisierung.
- `setShowSteps(true)` aktiviert die Schrittweise Abarbeitung des Algorithmus.
- `showGraph` läuft zufällig den Graphen ab. Nutzt dies als Beispiel für die Visualisierungsfunktionen.
- `node.status = Node.WHITE` färbt einen Knoten weiss (oder auch Node.BLACK, Node.GREY) ein.
- `resetState()` färbt alle Knoten weiss.

1.6. Breitensuche Implementieren (50 Punkte) Implementiert nun analog zur Tiefensuche die Breitensuche im vorgegebenen Methodenrumpf `DiGraph.breadthFirstSearch(Node startNode)`. Die Methode soll eine Liste mit Knoten zurückgeben, die so geordnet sind, wie euer Algorithmus diese Knoten besucht hat.

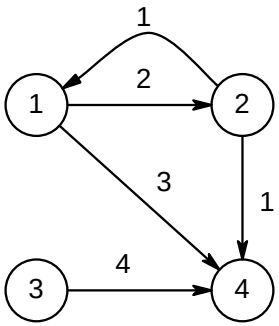


Abbildung 1: **Graph 01**

2. Aufgabe: Allgemeines

2.1. (Tut) Darstellung: Diskutiert die Darstellung des Graphen 01 und wie ein solcher Graph im Rechner repräsentiert werden könnte.

2.2. (Tut) Implementierung Im weiteren Verlauf der übung soll das Hinzufügen von Knoten und Kanten des Graphen implementiert werden. Für die Implementierung der Graphen ist folgendes Interface gegeben:

```
1  package graph;
2
3  // genaue Beschreibung der Methoden siehe Vorgaben.
4
5  public interface Graph {
6
7          public static final int WEIGHT_NO_EDGE = 0;
8
9          public Node addNode();
10
11         public void addEdge(Node startNode, Node targetNode, int weig
12
13         public List<Node> getNodes();
14
15         public List<Node> getAdjacentNodes(Node startNode);
16
17         public boolean isStopped();
18
19         public void setStopped(boolean status);
20
21         public List<Node> breadthFirstSearch(Node startNode);
22
23         public List<Node> breadthFirstSearch(int startNodeID);
24
25         public List<Node> depthFirstSearch(Node startNode);
26
27         public List<Node> depthFirstSearch(int startNodeID);
28
29         public boolean hasCycle();
30
31         public List<Node> topSort();
32
33         // ... further methods used for visualization, IO and homework assign
34
35 }
```


2.3. (Tut) Knoten hinzufügen Zur Darstellung der Knoten existiert die Klasse `Node`, für die Kanten wurde die Klasse `Edge` erstellt. Schreibt nun in der Klasse `DiGraph` die Methode `public Node addNode()`, die einen Knoten zum Graphen hinzufügt. Der Rückgabewert ist der erstellte Knoten, der als ID eine fortlaufende Zahl erhalten soll.

2.4. (Tut) Kanten hinzufügen Schreibt nun in der Klasse `Node` die Methode `public void addEdge(Node startnode, Node targetnode, int weight)`. Beachtet, dass es sich beim `DiGraph` um einen gerichteten Graphen handelt.

2.5. (Tut) Implementierung von getAdjacentNodes und isConnected Implementiert in der Klasse `DiGraph` die beiden Methoden `getAdjacentNodes(Node)` und `isConnected(Node, Node)` wie im Interface `Graph` als Kommentar angegeben. `getAdjacentNodes(Node)` soll eine Liste der direkt verbundenen Knoten zurückgeben (Richtung beachten) und `isConnected(Node startnode, Node endnode)` gibt an, ob es eine gerichtete Kante vom erstgenannten (startnode) zum zweiten genannten Knoten (endnode) gibt.