TU Berlin, Fakultät IV
Blankertz/Röhr/Stahl
MAR 4-3 und MAR 5-1

Aufgabenblatt 6

Graphenalgorithmen

Abgabe (bis 11.06.2019 19:59 Uhr)

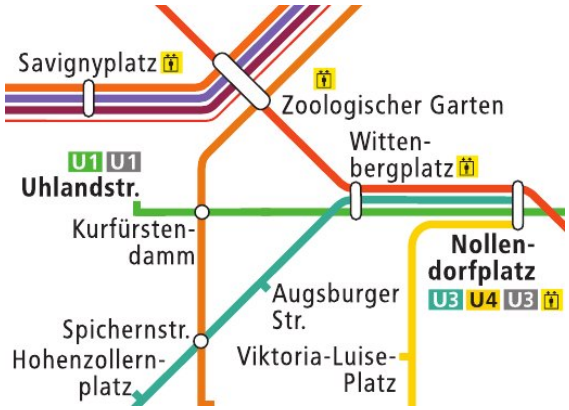
Die folgenden Dateien werden für die Bepunktung berücksichtigt:

Geforderte Dateien:	
`Blatt06/src/Maze.java`	Aufgabe 3.3, 3.5 & 3.6
`Blatt06/src/DepthFirstPaths.java`	Aufgabe 3.1 & 3.2
`Blatt06/src/RandomDepthFirstPaths.java`	Aufgabe 3.4

Als Abgabe wird jeweils nur die letzte Version im git gewertet.

Aufgabe 1: Graphenalgorithmen (Tut)

Betrachten Sie diesen U-Bahnplan:



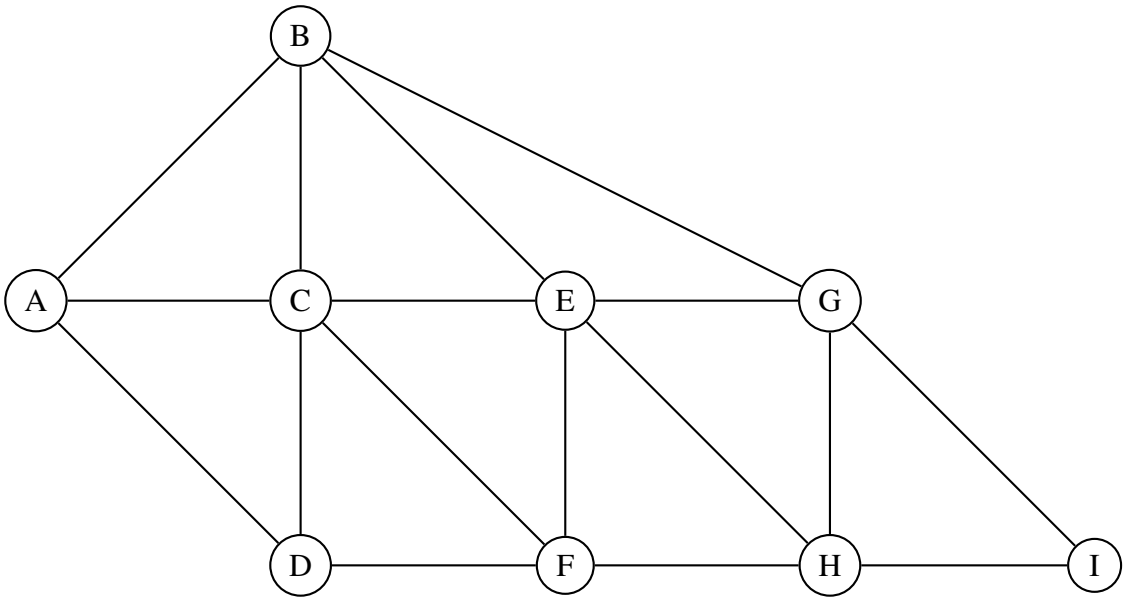
1.1 Auf welche Art und Weisen können sie diesen U-Bahnplan als Graph darstellen. Wie könnte man den Graphen in einem Inputfile darstellen?

1.2 Wie könnte man die Klasse Graph implementieren? Betrachten Sie unterschiedliche Lösungen und machen Sie sich deren Vor- und Nachteile klar.

1.3 Wie funktionieren die Breiten- und die Tiefensuche? Worin bestehen die Unterschiede?

1.4 Besprechen Sie die Vor- und Nachteile die benachbarten Knoten in einer LinkedList bzw. als Iterable zu speichern. (In Ihrer Hausaufgabe ist die Implementierung mit einer LinkedList gewählt.)

Aufgabe 2: BFS und DFS (Tut)



2.1 Führen Sie die Breitensuche auf dem gegebenen Graphen aus. Fangen Sie bei Knoten A an und notieren Sie alle Knoten in der Reihenfolge, in der sie von der Breitensuche in die Warteschlange geschrieben werden.
Gehen Sie dabei davon aus, dass in jedem Knoten die benachbarten Knoten in alphabetischer Reihenfolge abgearbeitet werden. Z.B. wird die Kante F-C vom Algorithmus vor der Kante F-D bearbeitet.

2.2 Führen Sie die Tiefensuche auf dem gegebenen Graphen aus. Fangen Sie bei Knoten A an und notieren Sie alle Knoten in der Reihenfolge, in der sie von der Tiefensuche entdeckt werden.
Gehen Sie dabei davon aus, dass in jedem Knoten die benachbarten Knoten in alphabetischer Reihenfolge abgearbeitet werden. Z.B. wird die Kante F-C vom Algorithmus vor der Kante F-D bearbeitet.

Aufgabe 3: Depth First Search

In dieser Aufgabe werden Sie mithilfe des DFS Algorithmus ein randomisiertes Labyrinth erstellen und lösen. Das Labyrinth wird repräsentiert von einem Graphen mit $V = N^2$ Knoten, die in einem quadratischen Gitter angeordnet sind (siehe Figure 1), und den dazwischen liegenden Kanten. Die Knoten sind hierbei die Kreuzungspunkte und die Kanten die direkten Verbindungen zwischen den Kreuzungspunkten. Die Kanten sind also die Wege im Labyrinth.
Hinweis:

- Schauen Sie sich den vorhandenen Code der Klassen, die Sie ergänzen müssen, sowie die Klasse `Graph` sorgfältig an.

3.1 Tiefensuche (18 Punkte)

Ergänzen Sie in der Klasse `DepthFirstPath` die Methoden

```
private void dfs(Graph G, int v)
```

und

```
public void nonrecursiveDFS(Graph G)
```

so, dass während der Ausführung bereits `postorder`, `preorder`, `edgeTo` und `distTo` berechnet werden. Dafür dürfen Sie die vorhandene Methode übernehmen und die entsprechenden Zeilen einfügen.
Hinweise:

- Nutzen Sie die gegebene Klasse `Graph`, um Ihren Code an einem einfachen Graphen zu testen. Erstellen Sie einen Graph durch das Erstellen eines Graphen mit n Knoten und das Hinzufügen von Kanten.
- Legen Sie sich ein einfaches Beispiel zurecht, bei dem Sie post- und preorder kennen. Lassen Sie sich alle Zwischenergebnisse ausgeben und überprüfen Sie, dass in beiden Varianten des Algorithmus das gleiche herauskommt.
- Die Klasse `In` können Sie ignorieren. Sollten Sie sich doch Input-Dateien für Graphen erstellen wollen: in dem File steht zuerst die Anzahl der Knoten, in der nächsten Zeile die Anzahl der Kanten und die restlichen Zeilen fangen mit einem Knoten an und alle weiteren Zahlen in der gleichen Zeile sind die benachbarten Knoten des Anfangsknotens.

3.2 Wege finden (13 Punkte)

Implementieren Sie die Methode

```
public List<Integer> pathTo(int v)
```

in der Klasse `DepthFirstPath`, sodass die Methode den Path rückwärts von `v` zum Ausgangspunkt (source) in der richtigen Reihenfolge zurückgibt.

Hinweise:

- Nutzen Sie `edgeTo` um den Weg rückwärts entlang zu gehen.
- Die gleiche Funktion brauchen Sie auch in `RandomDepthFirstPaths`, da Sie die Methode hier implementieren sollen, ist sie dort leer. **Kopieren Sie also Ihre Methode**, nachdem Sie sich sicher sind, dass sie funktioniert, in die Klasse `RandomDepthFirstPaths`.

3.3 Ausgangsgraph (15 Punkte)

Implementieren in der Klasse `Maze` die Methode

```
public Graph mazegrid()
```

Diese gibt einen Graphen der in Figure 1 gezeigten Struktur zurück.

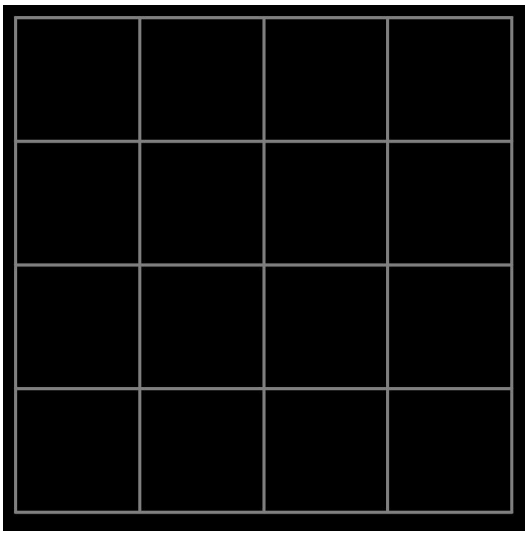


Figure 1: mazegrid()

Es ist also ein Graph, in dem alle Knoten in einem quadratischen Netz angeordnet und nur mit ihren unmittelbaren Nachbarn verbunden sind. Der Graph G hat die gleiche Anzahl an Knoten, wie der Graph `M` der Klasse.

3.4 Randomisierung (15+10* Punkte)

Implementieren Sie die Funktionen

```
private void randomDFS(Graph G, int v)
```

und als *Bonusaufgabe*

```
public void randomNonrecursiveDFS(Graph G)
```

in der Klasse `RandomDepthFirstPaths`. Dies sind die entscheidenden Funktionen, um ein Labyrinth mit DFS zu erstellen. Sie können sich bei beiden Funktionen an die Implementierung aus (3.1) halten. Bei dem randomisierten DFS Algorithmus wird statt des nächsten benachbarten

Knoten immer ein zufälliger benachbarter Knoten als nächstes markiert und bearbeitet.
Hinweise:

- Beachten Sie, dass `G.adj(v)` eine `List` ist und Sie somit Methoden aus `java.util.Collections` nutzen können, um die Randomisierung vorzunehmen.
- Die Methode `randomNonrecursiveDFS()` (Bonusaufgabe) wird nicht weiter benötigt. Sie dient nur zu Übungszwecken. In der Praxis könnte sie bei sehr großen Graphen benötigt werden, da die Rekursionstiefe in Java beschränkt ist (und die nicht-rekusive Variante effizienter ist).
- Für Bäume (Spezialfall von Graphen) lässt sich die nicht-rekursive Version der Tiefensuche einfacher implementieren (siehe die entsprechende Branch-and-Bound Variante in der Vorlesung). Für allgemeine Bäume funktioniert das nicht und es ist etwas mehr Aufwand, wie in der hier gegebenen Methode, notwendig.

3.5 Kanten hinzufügen (10 Punkte)

Implementieren Sie die Methoden

```
public boolean hasEdge(int v, int w)
```

und

```
public void addEdge(int v, int w)
```

in der Klasse `Maze`.
Die Methode `hasEdge` überprüft, ob die Kante zwischen `v` und `w` bereits im Graphen `M` (Objektvariable von `Maze`) enthalten ist. Lassen Sie keine reflexiven Kanten zu, also Kanten die von einem Knoten zu sich selbst gehen. Geben Sie dafür bei der Methode `hasEdge` für diese Kanten `true` zurück. Die Klasse `addEdge` soll eine Kante zum Graphen `M` hinzufügen.

3.6 Labyrinth (30 Punkte)

Implementieren Sie die Methoden

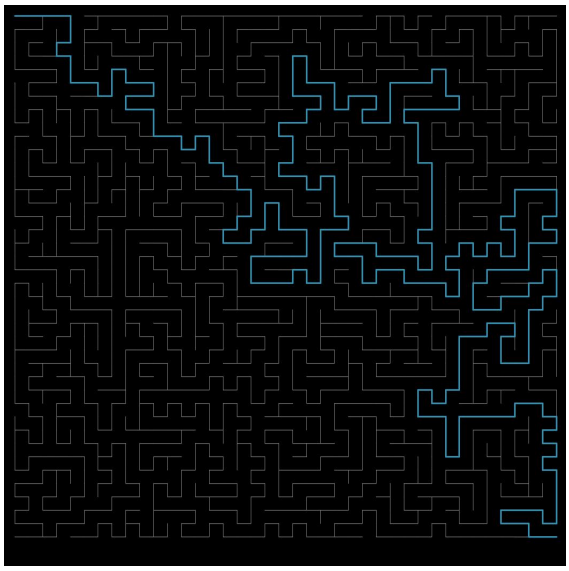
```
private void buildMaze()
```

und

```
public List<Integer> findWay(int v, int w)
```

in der Klasse `Maze`.
In `buildmaze` müssen zunächst alle Kanten/Pfade gefunden werden, die zu dem Labyrinth gehören. Dazu sollte ein randomisierten DFS auf einem Graphen G, der die Form aus Aufgabenteil 3.3 hat, benutzt werden. Diese Kanten werden dem Graphen `M` hinzugefügt. Es gibt in `M` keine doppelten Kanten.
Die Methode `findWay` sollte einen Pfad vom Knoten `v` nach `w` auf dem Graphen `M` in der richtigen Reihenfolge zurückgeben.

In der Visualisierung sähe das z.B. so aus:



Hinweise:

- Um einen Graphen zu visualisieren, können Sie die Klasse `GridGraph` nutzen. Sie plotten einen Graphen `M` mit `GridGraph vis= new GridGraph(M)` und einen Pfad `P` in einem Graphen `M` mit `GridGraph vis= new GridGraph(M, P)`. Nutzen Sie auch die Visualisierung, um Ihren Code zu überprüfen.
- Um einen *kürzesten* Weg im Labyrinth zu finden, benutzt man Breitensuche. Hier geben wir uns mit einem längeren Weg per Tiefensuche zufrieden, um die Implementation einer weiteren Methode einzusparen.