TU Berlin, Fakultät IV
Brock/Pfisterer/Koenig
MAR 5-1

Aufgabenblatt 6

Graphenalgorithmen

Wichtige Ankündigungen

- Es gibt drei Aufgabentypen:
  - **Tutorium** → Besprechung im Tutorium
  - **Hausaufgabe** → Eigenarbeit mit Hilfe in Rechnerübungen
  - **Klausurvorbereitung** → Optionale Klausurübung

Abgabe (bis 23.06.2025 23:59 Uhr)

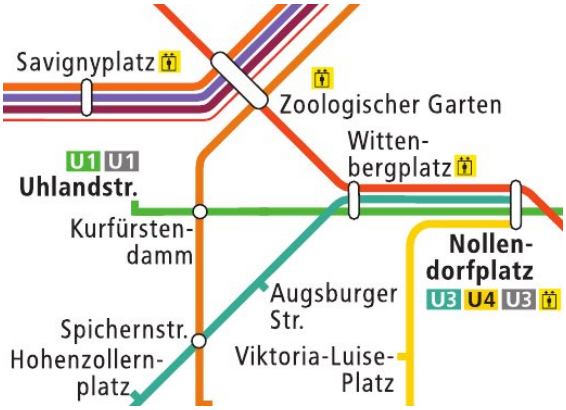
Die folgenden Dateien werden für die Bepunktung berücksichtigt:

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 main branch in git gewertet.

Aufgabe 1: Graphenalgorithmen (Tutorium)

Betrachten Sie diesen U-Bahnplan:

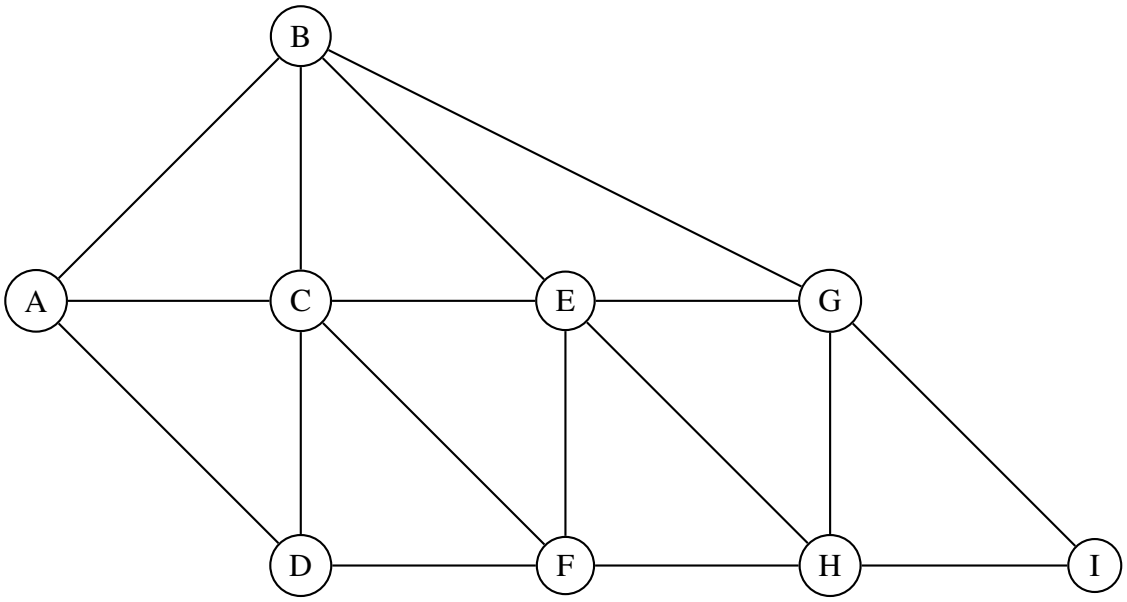


1.1 Auf welche Art und Weisen können Sie diesen U-Bahnplan als Graph darstellen. Wie könnte man den Graphen in einem Inputfile darstellen?

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?

Aufgabe 2: BFS und DFS (Tutorium)



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. Zum Beispiel 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. Zum Beispiel wird die Kante *F*-*C* vom Algorithmus vor der Kante *F*-*D* bearbeitet.

Aufgabe 3: Depth First Search (Hausaufgabe)

In dieser Aufgabe werden Sie mithilfe des DFS Algorithmus ein randomisiertes Labyrinth erstellen und lösen. Das Labyrinth wird von einem Graphen mit $V = N^2$ Knoten, die in einem quadratischen Gitter angeordnet sind (siehe Abbildung 1), und den dazwischen liegenden Kanten repräsentiert. 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 `DepthFirstPaths` die Methoden

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

und

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

In `dfs` soll die Tiefensuche vom Knoten `v` aus gestartet werden, in `nonrecursiveDFS` von Knoten `s` (eine Klassenvariable) aus. In beiden Methoden sollen `preorder`, `postorder`, `edgeTo` und `distTo` berechnet und in den entsprechenden Klassenvariablen gespeichert werden. Dafür dürfen Sie die vorhandene Methode übernehmen und die entsprechenden Zeilen einfügen.
Hinweise:

- Die Knoten können über ihre Indizes identifiziert werden. Nutzen Sie diese Indizes, um die oben angegebenen Listen bzw. Arrays zu füllen.
- in `edgeTo[w]` soll der Index des Knotens gespeichert werden, von welchem aus man zum Knoten mit Index `w` gelangt, wenn man die DFS ausführt. Beispiel: gegeben ein Graph mit zwei Knoten. Die Knoten tragen die Indizes 1 und 2 und sind mit einer Kante verbunden. Bei einer DFS startend bei Knoten 1 wird entsprechend Knoten 2 von Knoten 1 aus erreicht. Daher ist `edgeTo[2] = 1`.
- Sie können das Array `marked` nutzen, um bereits besuchte Knoten zu identifizieren.
- 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 (12 Punkte)

Implementieren Sie die Methode

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

in der Klasse `DepthFirstPath`, sodass die Methode den Pfad von `v` zum Ausgangspunkt (source, `s`) in der richtigen Reihenfolge als Liste zurückgibt oder `null`, falls ein solcher Pfad nicht existiert.

Hinweise:

- Nutzen Sie `edgeTo` um den Weg 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 Sie in der Klasse `Maze` die Methode

```
public Graph mazegrid()
```

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

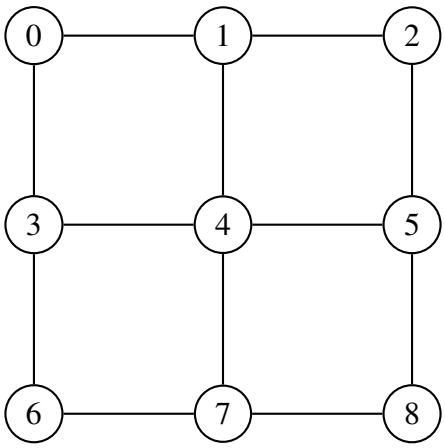


Abbildung 1: mazegrid, hier ein Beispiel für ein 3x3-Grid

Es ist also ein Graph, in dem alle Knoten in einem quadratischen Netz angeordnet und nur mit ihren unmittelbaren Nachbarn verbunden sind. Der zu erzeugende Graph G hat die gleiche Anzahl an Knoten, wie der Graph `M` der Klasse. Der Graph `M` enthält allerdings noch keine Kanten. Der Graph `G` soll die Kanten, wie oben beschrieben, enthalten.

3.4 Randomisierung (15 Punkte)

Implementieren Sie die Funktionen

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

und

```
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()` könnte in der Praxis z.B. bei sehr großen Graphen benötigt werden, da die Rekursionstiefe in Java beschränkt ist (und die nicht-rekusive Variante effizienter ist).

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)

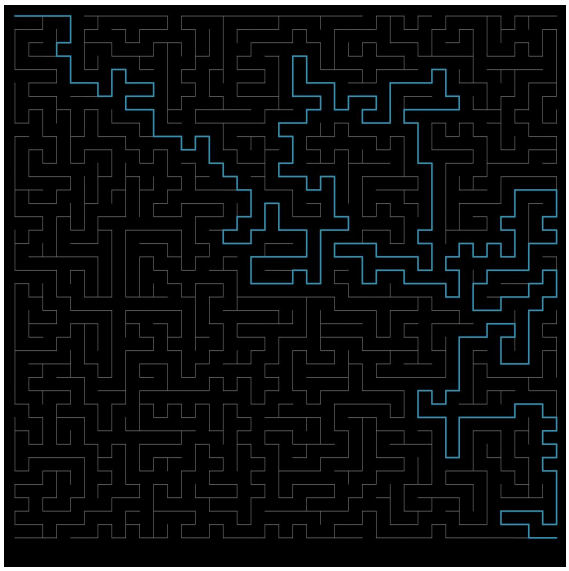
Implementieren Sie die Methoden

```
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 eine randomisierte 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 die Breitensuche. Hier geben wir uns mit einem längeren Weg per Tiefensuche zufrieden, um die Implementation einer weiteren Methode einzusparen.

Was Sie nach diesem Blatt wissen sollten:

- Was (ungerichtete und gerichtete) Graphen sind und welche Repräsentationen von Graphen es gibt.
- Was der Zweck einer Breiten- bzw. einer Tiefensuche ist.
- Wie die Breiten- und die Tiefensuche auf Graphen funktionieren
- Was die Breiten- und die Tiefensuche unterscheidet
- Wie man mit der Breiten- bzw. Tiefensuche einen Zyklus im Graphen finden kann und wie man Zusammenhangskomponenten erkennen kann
- Was die Haupt- und die Nebenreihenfolge sind
- Was topologische Sortierung ist