

Artemis 6.4.0

Course Overview

ge64baw

Courses > Praktikum: Grundlagen der Programmierung WS22/23 > Exercises > W06H03 - Fun with Graphs

W06H03 - Fun with Graphs

Hausaufgabe

Hard

Submission due: 8 months ago

Complaint due: 8 months ago

Points: 5 of 6

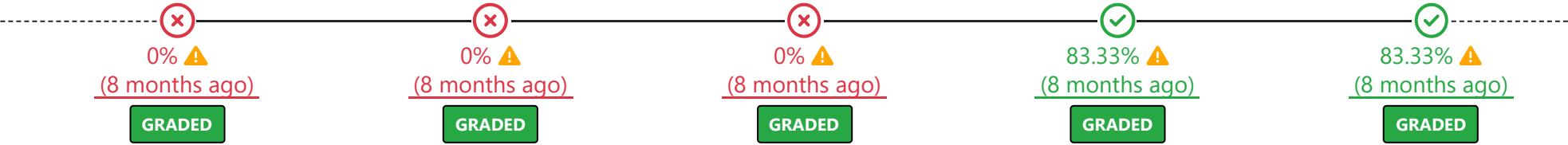
Assessment: automatic ?

Resume practice in exercise

83.33% (8 months ago)

GRADED

Recent results:



Show all results

Tasks:

Fun with Graphs

Obwohl der neue Super-Rechner der PUM noch nicht fertig gebaut ist, überlegen die Forschuine schon jetzt, was sie alles in Zukunft ausprobieren wollen. Sehr beliebt sind dabei Graphen und die unzähligen Algorithmen, die man auf ihnen ausführen kann. In dieser Aufgabe willst du ihnen ein paar Prototypen zeigen, wie du Graphen implementieren würdest.

Wichtig: In dieser Aufgabe lassen wir dir "künstlerische" Freiheit, wie du die geforderte Aufgabe umsetzt. Wichtig ist nur, dass du auch wirklich alle Anforderungen erfüllst.

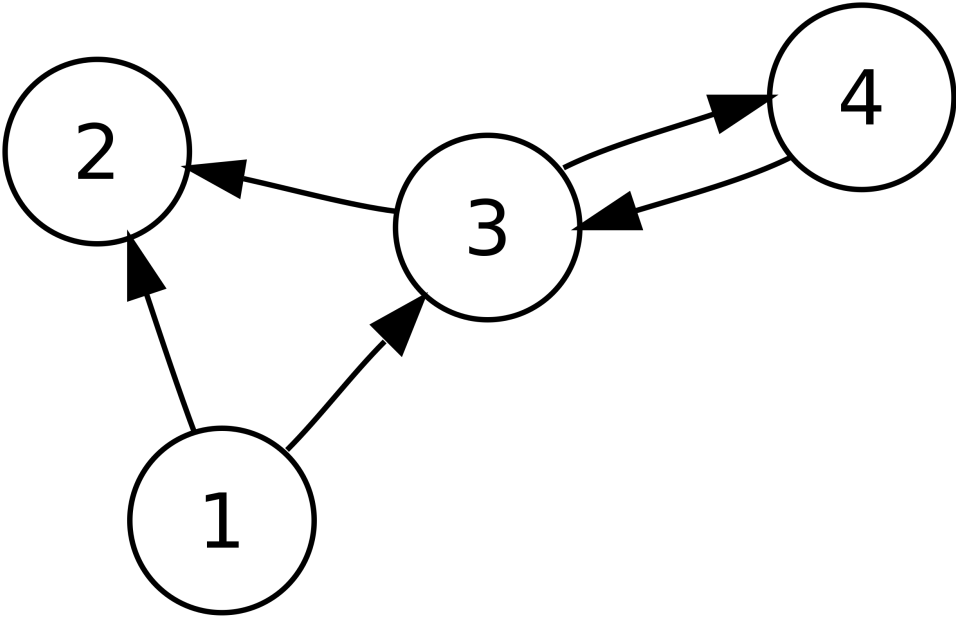
Hinweis: Parameter von Konstruktoren und Methoden folgen den in [diesem Zulip-Post](#) aufgelisteten Regeln. Insbesondere können also `int`-Parameter jeden möglichen Wert, der durch Integer dargestellt werden kann, annehmen.

Graphen

Was sind (gerichtete) Graphen?

Ein Graph besteht aus Knoten und Kanten. Stell dir am besten das ÖPNV-Netz vor. Die verschiedenen Haltestellen entsprechen den Knoten eines Graphen. Zwischen zwei Haltestellen existiert eine Kante genau dann, wenn ein Bus von einer zur anderen Haltestelle fährt. In dieser Aufgabe fokussieren wir uns ausschließlich auf gerichtete Graphen, das bedeutet man kann entlang einer Kante nur in eine Richtung gehen, aber nicht zurück.

Sieh dir dazu folgendes Beispiel an:



```
graph TD; 1((1)) --> 2((2)); 1((1)) --> 3((3)); 2((2)) --> 3((3)); 3((3)) --> 4((4)); 4((4)) --> 3((3));
```

Quelle

In diesem Netz kannst du von der 1 direkt zur 3 gehen, in die entgegengesetzte Richtung jedoch nicht. Um in beide Richtungen gehen zu können, muss der Graph sowohl eine Hin- als auch eine Rückkante enthalten, wie z.B. zwischen 3 und 4.

Verschiedene Implementierungen

Um Netzwerke zu speichern, gibt es viele Möglichkeiten, diese zu modellieren. Einige wenige davon werden dir hier erklärt. Aus diesen darfst du dann auch wählen, um diese Aufgabe zu lösen. Überlege dir dabei jeweils, welches Modell sinnvoll für die jeweilige Teilaufgabe ist. Hast du schon forgeschrittenes Wissen über Graphen, kannst du selbstverständlich auch andere Varianten wählen (die hier nicht aufgelistet sind). Solange du die Anforderungen erfüllst, ist hier alles erlaubt.

- Objektorientiert:
Als eifrige Java-Pinguine lieben wir die Objektorientierte Programmierung. Deshalb wollen wir unser Graphen als aller erstes mit Objekten

modellieren. Um unser Netzwerk zu modellieren, brauchen wir zunächst Objekte für unsere Knoten. Kanten zwischen den Knoten können nun z.B. wiederum durch Objekte organisiert werden. Einfacher ist es jedoch, wenn ein Knoten selbst seine Nachbarn abspeichert, entweder als Referenzen auf die jeweiligen Knoten-Objekte der Nachbarn oder einfacher noch über die ID des Knoten. Letzteres kann man z.B. mithilfe der von uns zur Verfügung gestellten Klasse `SimpleSet` machen. Wie du in der Aufgabe sehen wirst, haben die Knoten einen Namen (ihre `id` startet dabei von 0, 1, 2 usw.).

Tipp: In diesem Modell speichern wir nur das, was wir wirklich brauchen. Denk aber daran, dass das Erstellen von Objekten Speicher verbraucht.

2. Adjazenz Matrix:

Sieh dir folgende Matrix an:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Das ist die Adjazenzmatrix die zu dem Netz aus dem Bild von zuvor passt. Jede 1 in der Matrix entspricht genau einer Kante aus dem Bild. Kannst du jetzt schon erraten wie? Jede Zeile & Spalte der Matrix entspricht einem Knoten. Ist in einer Zeile i der j -te Eintrag eine 1 (true?), also $A_{ij} = 1$, dann existiert eine Kante von i nach j .

Tipp 1: Wir Info-Pinguine lieben Matrizen, weil man sie so schön in zwei dimensional Arrays abspeichern kann.

Tipp 2: Die Größe der Matrix steigt also quadratisch zur Anzahl der Knoten des Graphen, ist aber unabhängig von der Anzahl der Kanten.

3. Inzidenz Matrix:

Sieh dir folgende Matrix an:

$$I = \begin{pmatrix} -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix}$$

Das ist die Inzidenzmatrix die zu dem Netz aus dem Bild von zuvor passt. Sie ist sehr ähnlich zur Adjazenzmatrix, hier entspricht jedoch jede Zeile einem Knoten während jede Spalte einer Kante zuzuordnen ist. Jeder Wert in einer Spalte der nicht 0 ist bedeutet, dass der Knoten der entsprechenden Zeile teil der Kante ist. Anhand des Vorzeichens erkennen wir die Richtung der Kante (von negativ, nach positiv). Eine Kante von i nach j haben wir also genau dann, wenn $I_{is} = -1$ und $I_{js} = 1$ ist (s ist dabei der Index einer Spalte - Kante).

Tipp: Mach dir Gedanken darüber was passiert, wenn du eine neue Kante zum Graphen hinzufügen möchtest.

Unser Template

In dem Code-Template, welches wir dir über Artemis zur Verfügung stellen, sind bereits einige Funktionalitäten implementiert, die du zum Lösen der Aufgabe verwenden kannst bzw. die wir zum Testen benötigen. Im Folgenden wird dir kurz erklärt, wie du damit umgehen sollst:

Die Klasse `SimpleSet`

`SimpleSet` ist eine Menge (von Integern), wie du sie aus der Mathematik kennst: Du kannst Zahlen hinzufügen (`add(int)`) und abfragen ob die Menge eine Zahl enthält (`contains(int)`). `toArray()` gibt dir ein Array bestehend aus allen Zahlen die in diesem `SimpleSet` gespeichert sind zurück. Intern verwendet `SimpleSet` Konstrukte, die du vermutlich noch nicht kennst/noch nicht kennen musst. Darum brauchst du dir aber keine Gedanken zu machen. Verwende ein `SimpleSet` einfach, indem du mit `new SimpleSet()` eine neue, erstmal leere Menge erzeugst und dann die drei eben beschriebenen Methoden darauf anwendest, um Zahlen hinzuzufügen, zu überprüfen, ob eine gegebene Zahl vorhanden ist bzw. ein Array aller enthaltenen Zahlen zu generieren.

Das Interface `Graph`

Java-Interfaces musst du zu diesem Zeitpunkt noch nicht verstehen. Verständnis von `Graph` ist nicht notwendig (oder hilfreich), um diese Aufgabe zu lösen. Wir benötigen `Graph` hauptsächlich für unsere Implementierung des im nächsten Punkt beschriebenen Dijkstra-Algorithmus. Ignoriere diese Datei sowie die Add-Ons `implements Graph` hinter `public class SparseGraph` und `public class DenseGraph` also gerne einfach.

Dijkstra

Im Template findest du die Klasse `Dijkstra`. Hierzu gibt es keine Aufgaben! Die Implementierung ist nur zum Spielen gedacht! Um diese Aufgabe zu lösen, kannst du `Dijkstra.java` also gerne auch einfach ignorieren, wir wollen aber zu Neugier ermutigen. Einfach nur Netze zu modellieren, wäre zu langweilig, deshalb hast du in dieser Klasse einen Algorithmus, der den kürzesten Pfad zwischen zwei Knoten bestimmen kann (Jede Kante hat bei unserem simplen Netz Länge 1, deshalb ist der kürzeste Pfad, der Weg vom Start zum Ziel mit der minimalen Anzahl von Knoten/Kanten). Der Dijkstra Algorithmus gehört zum kleinen 1x1 der Informatik und wird dir noch in mehreren Vorlesungen begegnen, aller spätestens kommenden Semester in "Grundlagen Algorithmen und Datenstrukturen". Wenn du deine Implementierungen einmal für Dijkstra nutzen möchtest, kannst du das wie folgt tun:

```
Graph g = new DenseGraph(10);           // oder SparseGraph
int start = 0;                          // die ID des Knoten von dem aus wir losgehen möchten
int target = 7;                         // die ID des Knoten den wir erreichen möchten
int[] path = Dijkstra.dijkstra(g, start, target); // der kürzeste Pfad
System.out.println(Arrays.toString(path)); // eine Ausgabe in der Konsole
```

Sparse Graphen

Einen Graphen nennen wir "sparse", wenn er nur sehr wenige Kanten enthält. Im Folgenden sollst du die Klasse `SparseGraph` implementieren. Diese Datenstruktur soll für das Speichern von sparse Graphen optimiert sein. Natürlich muss die Datenstruktur dennoch DAU-safe (Dümmster anzunehmender User) sein, d.h. es soll dennoch möglich sein, beliebige Graphen in der Datenstruktur zu speichern. Überlege dir also, welche der obigen

Implementierungen auch für Graphen mit mehreren Millionen Knoten, aber nur einigen wenigen Kanten pro Knoten, noch effizient (bzgl. Laufzeit der Methoden und Speicherverbrauch) ist.

`SparseGraph` implementiert das Interface `Graph`. Du kannst dir `Graph` gerne schon für die kommende Woche ansehen, jetzt ist es aber noch nicht notwendig zu verstehen was ein Java-Interface ist.

1. ? **Konstruktor** No results

Der Konstruktor erwartet `nodes` (`int`), die Anzahl der Knoten des Graphen. Die Knoten werden im Folgenden über ihre "id"/"Namen" betitelt. Dazu sind die Knoten nummeriert (`0 - nodes-1`). Passe den Konstruktor für deine Implementierung entsprechend an.

2. ? **getNumberOfNodes** No results

Die Methode `getNumberOfNodes()` soll die Anzahl der Knoten des Graphen zurückgeben. Dem Konstruktor wurde diese Zahl bei der Initialisierung übergeben (`nodes`).

3. ? **addEdge** No results

Die Methode `addEdge(int, int)` erwartet die beiden `int`-Werte `from` und `to`. Dabei handelt es sich um zwei Knoten, zwischen denen wir eine Kante hinzufügen möchten. Enthält der `SparseGraph` noch keine gerichtete Kante von `from` nach `to`, soll diese nun erstellt werden.

4. ? **isAdj** No results

Die Methode `isAdj(int, int)` erwartet die beiden `int`-Werte `from` und `to` und soll einen `boolean` zurückgeben. Dabei handelt es sich um zwei Knoten. Die Methode soll `true` returnen, wenn eine Kante von `from` nach `to` existiert, andernfalls `false`.

5. ? **getAdj** No results

Die Methode `getAdj(int)` erwartet den `int`-Wert `id` (ein Knoten des Graphen). Existiert dieser Knoten im `SparseGraph`, soll ein Array aller benachbarten Knoten zurückgegeben werden (Array aus Knoten-IDs). Ein Knoten gilt als benachbart, wenn es eine gerichtete Kante von `id` zu diesem anderen Knoten gibt. Ist der Knoten `id` nicht im Graph enthalten, soll die Methode `null` zurück geben.

Public Test Summary

Hier werden dir die Ergebnisse der Public Tests zusammenfassend angezeigt. Um auf die Teilaufgabe "Sparse Graphen" Punkte zu bekommen, müssen diese alle passen. Die Punkte gibt's dann auf die drei Methoden `addEdge()`, `isAdj()` und `getAdj()`, wenn die Public Tests durchlaufen und die gewählte Implementierung von `SparseGraph` eine für sparse Graphen effiziente ist.

? **Public Tests** No results

Testet deine Abgabe nach jedem Push neu.

Optimiert für Sparsity

Hier werden dir die Ergebnisse der automatischen Tests, ob korrekt auf Sparsity optimiert wurde, nach der Deadline angezeigt:

? **Hidden Tests** No results

Testet deine Abgabe nach der Deadline.

Dense Graphen

Einen Graphen nennen wir "dense", wenn er sehr viele Kanten enthält. Im Folgenden sollst du die Klasse `DenseGraph` implementieren. Diese Datenstruktur soll für das Speichern von dense Graphen optimiert sein. Natürlich muss die Datenstruktur dennoch DAU-safe (Dümmster anzunehmender User) sein, d.h. es soll dennoch möglich sein, beliebige Graphen in der Datenstruktur zu speichern. Überlege dir also, welche der obigen Implementierungen auch für Graphen mit nicht ganz so vielen Knoten (vllt. einige zehntausend), aber dafür von jedem Knoten aus Kanten zu einem großen Anteil der anderen Knoten, noch effizient (bzgl. Laufzeit der Methoden und Speicherverbrauch) ist.

`DenseGraph` implementiert das Interface `Graph`. Du kannst dir `Graph` gerne schon für die kommende Woche ansehen, jetzt ist es aber noch nicht notwendig zu verstehen was ein Java-Interface ist.

1. ? **Konstruktor** No results

Der Konstruktor erwartet `nodes` (`int`), die Anzahl der Knoten des Graphen. Die Knoten werden im Folgenden über ihre "id"/"Namen" betitelt. Dazu sind die Knoten nummeriert (`0 - nodes-1`). Passe den Konstruktor für deine Implementierung entsprechend an.

2. ? **getNumberOfNodes** No results

Die Methode `getNumberOfNodes()` soll die Anzahl der Knoten des Graphen zurückgeben. Dem Konstruktor wurde diese Zahl bei der Initialisierung übergeben (`nodes`).

3. ? **addEdge** No results

Die Methode `addEdge(int, int)` erwartet die beiden `int`-Werte `from` und `to`. Dabei handelt es sich um zwei Knoten, zwischen denen wir eine Kante hinzufügen möchten. Enthält der `DenseGraph` noch keine gerichtete Kante von `from` nach `to`, soll diese nun erstellt werden.

4. ? **isAdj** No results

Die Methode `isAdj(int, int)` erwartet die beiden `int`-Werte `from` und `to` und soll einen `boolean` zurückgeben. Dabei handelt es sich um zwei Knoten. Die Methode soll `true` zurückgeben, wenn eine Kante von `from` nach `to` existiert, andernfalls `false`.

5. ? **getAdj** No results

Die Methode `getAdj(int)` erwartet den `int`-Wert `id` (ein Knoten des Graphen). Existiert dieser Knoten im `DenseGraph`, soll ein Array aller benachbarten Knoten zurückgegeben werden (Array aus Knoten-IDs). Ein Knoten gilt als benachbart, wenn es eine gerichtete Kante von `id` zu diesem anderen Knoten gibt. Ist der Knoten `id` nicht im Graph enthalten, soll die Methode `null` zurückgeben.

Public Test Summary

Hier werden dir die Ergebnisse der Public Tests zusammenfassend angezeigt. Um auf die Teilaufgabe "Dense Graphen" Punkte zu bekommen, müssen diese alle passen. Die Punkte gibt's dann auf die drei Methoden `addEdge()`, `isAdj()` und `getAdj()`, wenn die Public Tests durchlaufen und die gewählte Implementierung von `DenseGraph` eine für dense Graphen effiziente ist.

? **Public Tests** No results

Testet deine Abgabe nach jedem Push neu.

Optimiert für Density

Hier werden dir die Ergebnisse der automatischen Tests, ob korrekt auf Density optimiert wurde, nach der Deadline angezeigt:

 **Hidden Tests** No results

Testet deine Abgabe nach der Deadline.

Beispiele & toGraphviz()

Beide von dir zu implementierenden Klassen bieten die Methode `toGraphviz()` an. Diese gibt einen `String` zurück (du musst dazu schon die Methode `getAdj(int)` implementiert haben). Diesen `String` kannst du auf [dieser Website](#) in das linke Textfeld kopieren. Auf der rechten Hälfte des Bildschirms solltest du jetzt einen Plot des Graphen sehen. Oben siehst du das Drop-Down-Menu `Engine`, hier kannst du verschiedene Optionen ausprobieren, damit der Graph leserlich repräsentiert wird (`circo` ist häufig eine gute Wahl :)).

Beispiel: Sparse Graph

[Sparse Graph](#)

20 Knoten; 30 Kanten -> viele Knoten, aber nicht viel mehr Kanten als Knoten

Beispiel: Dense Graph

[Dense Graph](#)

10 Knoten; 75 Kanten -> nicht so viele Knoten, dafür aber viele Kanten pro Knoten

Viel Erfolg!!!

[Lösungsvorschlag \(nach der Deadline\)](#)

[Tests \(nach der Deadline\)](#)

Exercise details

| | |
|-----------------|--------------------|
| Release date: | Nov 24, 2022 18:30 |
| Submission due: | Dec 11, 2022 18:00 |
| Complaint due: | Dec 18, 2022 18:00 |

Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have **998** complaints left. 