Karlsruher Institut für Technologie

Programmieren – Sommersemester 2015
Software-Design und Qualität (SDQ)
`https://sdqweb.ipd.kit.edu/wiki/Programmieren`
Prof. Dr. Ralf H. Reussner · Philipp Merkle · Kiana Rostami

Abschlussaufgabe 2

Ausgabe: 03.08.2015 – 13:00
Abgabe: 31.08.2015 – 13:00

Bearbeitungshinweise

- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen[1]
- Programmcode muss in englischer Sprache verfasst sein
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute
- Verwenden Sie keine Klassen der Java-Bibliotheken ausgenommen Klassen der Pakete `java.lang` und `java.io`, es sei denn die Aufgabenstellung erlaubt ausdrücklich weitere Pakete[1]
- Achten Sie auf fehlerfrei kompilierenden Programmcode[1]
- Halten Sie alle Whitespace-Regeln ein[1]
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen[1]
- Halten Sie die Regeln zu Javadoc-Dokumentation ein[1]
- Nutzen Sie nicht das default-Package[1]
- Halten Sie auch alle anderen Checkstyle-Regeln ein

Abgabemodalitäten

Die Praktomat-Abgabe wird am **Montag, den 17. August**, freigeschaltet. Geben Sie die Java-Klassen als `*.java`-Dateien ab. Laden Sie die Terminal-Klasse nicht mit hoch.

Planen Sie für die Abgabe ausreichend Zeit ein, sollte der Praktomat Ihre Abgabe wegen einer Regelverletzung ablehnen.

Fehlerbehandlung

Ihre Programme sollen auf ungültige Benutzereingaben mit einer aussagekräftigen Fehlermeldung reagieren. Aus technischen Gründen muss eine Fehlermeldung unbedingt mit `Error,` beginnen. Eine Fehlermeldung führt nicht dazu, dass das Programm beendet wird; es sei denn, die nachfolgende Aufgabenstellung verlangt dies ausdrücklich. Achten Sie inbesondere auch darauf, dass unbehandelte `RuntimeExceptions`, bzw. Subklassen davon—sogenannte *Unchecked Exceptions*—nicht zum Abbruch des Programms führen.

[1] Der Praktomat wird die Abgabe zurückweisen, falls diese Regel verletzt ist.

Fluginformationssystem (20 Punkte)

Betrachten Sie zum Einstieg den in Abbildung 1 dargestellten Graphen. Dieser Graph beschreibt Flughäfen (dargestellt als Knoten) und Flugrouten zwischen Flughäfen (dargestellt als Kanten). Jeder Kante bzw. Flugroute ist eine Zahl zugeordnet, welche die Distanz zwischen beiden Flughäfen ausdrückt. Auf Basis dieser Daten lassen sich unter anderem folgende Fragestellungen beantworten:

- Gibt es einen Non-Stop-Flug zwischen zwei bestimmten Flughäfen?
- Falls nein, wie viele Zwischenlandungen enthält die kürzeste Route?
- Welche Flugdistanz wird dabei zurückgelegt?

In dieser Aufgabe programmieren Sie ein vereinfachtes Fluginformationssystem, das seinen Benutzern Fragestellungen dieser Art beantwortet.

Sie dürfen für Ihre Implementierung das `java.util`-Paket nutzen.

Lesen Sie die in den Abschnitten A, C und D vorgestellten Datenstrukturen und Algorithmen aufmerksam durch. Sie benötigen diese Grundlagen für Ihre Implementierung.

A Grundlagen: Graph

Ein *Graph* besteht aus *Knoten*, die durch *Kanten* miteinander verbunden sein können: Bei einem *gerichteten Graph* besitzen die Kanten außerdem eine Richtung: jede Kante zeigt von einem Startknoten auf einen Zielknoten. Wenn zwischen zwei Knoten höchstens eine Kante je Richtung erlaubt ist, spricht man von einem *Graph ohne Mehrfachkanten*.

Formeller ausgedrückt ist ein Graph G ein geordnetes Paar (V, E), wobei V die Menge seiner Knoten (*Vertices*) und E die Menge seiner Kanten (*Edges*) bezeichnet. Eine Kante $e \in E$ ist ein geordnetes Paar (v_s, v_t) mit $v_s \in V$ und $v_t \in V$. Dabei bezeichnet v_s den Startknoten und v_t den Zielknoten der Kante e.

B Flugrouten-Graph

Ein *Flugrouten-Graph* ist ein gerichteter Graph $G = (V, E)$ ohne Mehrfachkanten.

Jeder Knoten $v \in V$ repräsentiert einen Flughafen. Ein Flughafen ist eindeutig durch seinen *IATA-Code* gekennzeichnet. Dieser besteht aus drei Buchstaben zwischen A und Z und ist durch den regulären Ausdruck `[A-Z]{3}`

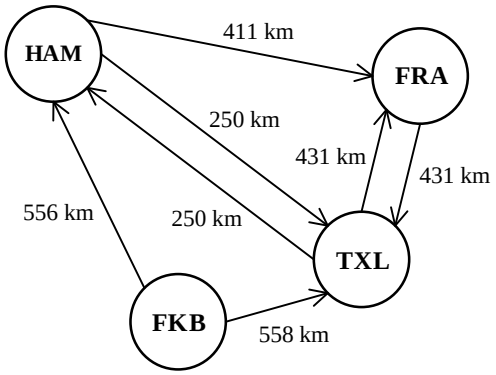

Abbildung 1: Beispiel eines Flugrouten-Graph. Knoten repräsentieren Flughäfen, mit IATA-Codes als Knotennamen. Kanten repräsentieren Flugrouten, mit Distanzen als Kantenbeschriftungen.

beschrieben. Dabei drückt die in geschweiften Klammern eingefasste Zahl 3 aus, dass genau drei Zeichen aus der Zeichenklasse `[A-Z]` erwartet werden. Weiterhin besitzt jeder Flughafen einen *Namen*.

Jede Kante $e = (v_s, v_t) \in E$ repräsentiert eine Flugverbindung. Dabei ist v_s der Startflughafen und v_t der Zielflughafen. Weiterhin existiert eine Funktion $distance : E \to \mathbb{N}$, die jeder Kante eine Flugdistanz zuordnet.

B.1 Serialisierung eines Flugrouten-Graphen

Unter einer *Serialisierung* verstehen wir im Rahmen dieser Aufgabe die textuelle Repräsentation eines Flugrouten-Graphen: ein Format, um einen solchen Graphen in eine Datei zu speichern bzw. aus einer Datei auszulesen.

Eine Serialisierung besteht aus zwei Bereichen. Der erste Bereich beschreibt die Knoten des Graphen. Der zweite Bereich beschreibt die Kanten des Graphen. Beide Bereiche sind durch eine Zeile mit dem Inhalt `--` voneinander separiert (vergleiche Abschnitt F.7).

Der **erste Bereich** enthält zwei oder mehrere Zeilen in beliebiger Reihenfolge. Jede Zeile beschreibt einen Flughafen (Knoten) und ist folgendermaßen aufgebaut:

```
<IATA-Code>;<Flughafen-Name>
```

`<IATA-Code>` ist ein String, der durch den regulären Ausdruck `[A-Z]{3}` beschrieben ist. Der IATA-Code dient zur eindeutigen Kennzeichnung eines Flughafens. `<Flughafen-Name>` ist ein String, der durch den regulären Ausdruck `[a-zA-Z_-]+` beschrieben ist und bezeichnet den Flughafen, welcher durch den IATA-Code identifiziert ist.

Der **zweite Bereich** enthält eine oder mehrere Zeilen in beliebiger Reihenfolge. Jede Zeile beschreibt eine Flugroute (Kante) und ist folgendermaßen aufgebaut:

```
<Startflughafen>;<Zielflughafen>;<Distanz>
```

Jede Flugroute verläuft von einem Startflughafen zu einem Zielflughafen, wobei gilt Startflughafen $\neq$ Zielflughafen. `<Startflughafen>` und `<Zielflughafen>` sind jeweils durch ihren IATA-Code repräsentiert, der genauso aufgebaut ist wie zuvor beschrieben. `<Distanz>` ist die Entfernung zwischen Start- und Zielflughafen in Kilometern, ausgedrückt als `Integer`-Zahl größer 0.

Eingabedatei für das in Abb. 1 dargestellte Beispiel

```
HAM;Hamburg
FKB;Karlsruhe_Baden-Baden
TXL;Berlin-Tegel
FRA;Frankfurt_am_Main
--
FKB;HAM;556
FKB;TXL;558
TXL;HAM;250
HAM;TXL;250
TXL;FRA;431
FRA;TXL;431
HAM;FRA;411
```

C Grundlagen: Adjazenz- und Distanzmatrix

C.1 Matrix

Eine *Matrix* ist eine tabellarische (d.h. 2-dimensionale) Anordnung von Elementen, in unserem Fall von ganzen Zahlen.

Der Eintrag $m_{i,j}$ bezeichnet die Zahl, die in der i-ten Zeile und j-ten Spalte einer Matrix M steht. Nachfolgend drücken wir dies in Kurzschreibweise aus: $M = (m_{i,j})$.

C.2 Adjazenzmatrix

Eine *Adjazenzmatrix* ist eine spezielle Matrix zur Repräsentation eines Graphen.

Schritt 1: Knoten durchnummerieren Bevor wir eine Adjazenzmatrix bilden können, müssen zunächst alle Knoten $v \in V$ fortlaufend durchnummeriert werden, beginnend mit 0. Eine gültige Nummerierung wäre beispielsweise:

$$\begin{aligned} \text{HAM} &\mapsto 0 \\ \text{FKB} &\mapsto 1 \\ \text{TXL} &\mapsto 2 \\ \text{FRA} &\mapsto 3 \end{aligned}$$

Diese Nummerierung erlaubt es uns, $(1, 0) \in E$ zu schreiben, um auszudrücken, dass eine Kante von FKB (mit der Nummer 1) nach HAM (mit der Nummer 0) verläuft. Beachten Sie, dass diese Nummerierung beliebig ist. Ebensogut hätten wir folgende Nummerierung wählen können: $\text{FKB} \mapsto 0,\ \text{FRA} \mapsto 1,\ \text{HAM} \mapsto 2,\ \text{TXL} \mapsto 3$.

Schritt 2: Adjazenzmatrix bilden Ein Graph $G = (V, E)$ lässt sich nun wie folgt als Adjazenzmatrix $A = (a_{i,j})$ darstellen, wobei die Anzahl der Zeilen bzw. Spalten in A der Knotenanzahl des Graphen entspricht:

$$a_{i,j} = \begin{cases} 1 & falls\ (i,j) \in E \\ 0 & sonst \end{cases}$$

Steht an der Stelle $a_{i,j}$ eine 1 bedeutet das somit, dass Knoten i mit Knoten j durch die Kante (i, j) verbunden ist. Ist $a_{i,j}$ hingegen 0, bedeutet das, dass die Knoten i und j nicht durch eine Kante verbunden sind.

Beispiel Der Graph aus Abbildung 1 lässt sich durch die Adjazenzmatrix A darstellen:

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

In Verbindung mit der oben eingeführten Beispiel-Nummerierung drückt der Eintrag $a_{1,2} = 1$ (dargestellt in Fettdruck) aus, dass ausgehend von Knoten FKB eine Kante auf Knoten TXL zeigt.

C.3 Distanzmatrix

Eine Adjazenzmatrix, welche die Entfernung zwischen jeweils zwei verbundenen Knoten ausdrückt, nennen wir *Distanzmatrix*.

Schritt 1: Knoten durchnummerieren siehe Abschnitt C.2

Schritt 2: Distanzmatrix bilden Die Distanzmatrix $D = (d_{i,j})$ für einen gerichteten Graphen $G = (V, E)$ ist folgendermaßen definiert:

$$d_{i,j} = \begin{cases} distance(e) & \text{falls } e \in E \text{ mit } e = (i,j) \\ \infty & \text{sonst} \end{cases}$$

Dabei liefert die Funktion $distance(e)$ die in der Kante e gespeicherte Distanz zurück.

Beispiel Der Graph aus Abbildung 1 lässt sich durch folgende Distanzmatrix D darstellen. Dabei nutzen wir wieder die in Abschnitt C.2 beschriebene Nummerierung.

$$D = \begin{pmatrix} \infty & \infty & 250 & 411 \\ 556 & \infty & 558 & \infty \\ 250 & \infty & \infty & 431 \\ \infty & \infty & 431 & \infty \end{pmatrix}$$

D Grundlagen: Graph-Algorithmen

D.1 Anzahl der Pfade zwischen zwei Knoten berechnen

Auf Basis einer **Adjazenzmatrix** (nicht der Distanzmatrix) lässt sich berechnen, wie viele Pfade von Knoten i zu Knoten j verlaufen, die eine bestimmte Pfadlänge besitzen. Pfade der Länge 1 sind in unserem Fall Non-Stop-Flüge ohne Zwischenlandung. Diese kann man direkt aus der Adjazenzmatrix ablesen. Pfade der Länge 2 sind in unserem Fall Flüge mit genau einer Zwischenlandung. Diese lassen sich mittels **Matrix-Multiplikation** bestimmen, indem die Adjazenzmatrix mit sich selbst multipliziert wird.

Für zwei quadratische Matrizen $A = (a_{i,j})$ und $B = (b_{i,j})$ erhält man die Ergebnismatrix $C = (c_{i,j})$ folgendermaßen:

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$

Dabei ist m die Anzahl der Zeilen bzw. Spalten von A und B. Weil A und B quadratisch sind, besitzt auch C m Zeilen und Spalten.

Sei A die Adjazenzmatrix aus Abschnitt C.2, dann enthält die aus der Multiplikation $A \times A$ resultierende Matrix C die Anzahl aller Pfade der Länge ~~1~~ 2 zwischen zwei bestimmten Knoten:

$$C = A \times A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 0 & 2 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Aus der Ergebnismatrix C können wir ablesen, dass es genau einen Pfad der Länge 2 gibt, der von FKB nach TXL führt (weil $c_{1,2} = 1$). Für die Route FKB nach FRA gibt es zwei Pfade der Länge 2 (weil $c_{1,3} = 2$). Die Ergebnismatrix beantwortet zwar nicht, *welche* Pfade dies sind, doch mit einem Blick auf den Graphen sind die beiden Pfade leicht zu entdecken: FKB → HAM → FRA, sowie FKB → TXL → FRA.

Statt $A \times A$ schreiben wir im Folgenden kürzer A^2. Statt $A \times A \times A$ schreiben wir A^3, und so weiter.

Berechnet man die Matrix A^l, kann man daraus die Anzahl aller Pfade ablesen, die genau die Länge l haben.

D.2 Kürzeste Pfade zwischen zwei Knoten berechnen

Die Länge des jeweils kürzesten Pfades zwischen allen Knotenpaaren in einem Graphen berechnet der **Floyd-Algorithmus**. Als Eingabe erwartet der Algorithmus die **Distanzmatrix** des Graphen G. Der Algorithmus berechnet dann alle kürzesten Pfade in G und speichert deren Länge in einer Matrix $S = (s_{i,j})$. Der Eintrag $s_{i,j}$ beinhaltet die Länge des kürzesten Pfades zwischen den Knoten i und j.

Algorithmus 1 beschreibt den Floyd-Algorithmus in Form von Pseudocode.

Anmerkung: Zur Umsetzung des Algorithmus ist es nicht notwendig, dessen genaue Funktionsweise zu verstehen.

Algorithmus 1 Floyd-Algorithmus zur Berechnung kürzester Pfade in einem Graphen $G = (V, E)$

```
1:  function ShortestPaths(D)                          // D is the distance matrix of G
2:      shortest ← D
3:      for k ← 0 to n − 1 do                          // n = |V|, i.e. the number of nodes in G
4:          for i ← 0 to n − 1 do
5:              for j ← 0 to n − 1 do
6:                  shortest[i, j] = min(shortest[i, j], shortest[i, k] + shortest[k, j])
7:              end for
8:          end for
9:      end for
10:     return shortest
11: end function
```

E Kommandozeilenargumente

Ihr Programm nimmt als erstes und einziges Kommandozeilenargument einen Pfad auf eine Textdatei entgegen. Diese Datei beschreibt einen Flugrouten-Graphen und ist nach dem in Abschnitt B.1 eingeführten Format aufgebaut.

Zum Einlesen einer Datei beachten Sie den Hinweis im Anhang.

Tritt beim Verarbeiten des Kommandozeilenarguments ein Fehler auf, so wird eine Fehlermeldung ausgegeben und das Programm beendet sich mittels `System.exit(1)`.

F Interaktive Benutzerschnittstelle

Nach dem Start nimmt Ihr Programm über die Konsole mittels `Terminal.readLine()` sechs Befehle entgegen. Nach Abarbeitung eines Befehls wartet das Programm auf weitere Befehle, bis das Programm irgendwann durch `quit` beendet wird.

Die nachfolgenden Befehle operieren auf dem Flugrouten-Graphen $G = (V, E)$, der beim Programmstart eingelesen wurde (vergleiche Abschnitt E).

F.1 `airport`-Befehl

Der `airport`-Befehl gibt den Namen eines bestimmten Flughafens auf die Konsole aus.

Eingabeformat `airport <Flughafen>` `<Flughafen>` ist ein IATA-Code wie in Abschnitt B beschrieben.

Ausgabeformat Falls `<Flughafen>` ein gültiger Knoten ist, wird der Name des Flughafens ausgegeben. Im Fehlerfall wird eine aussagekräftige Fehlermeldung ausgegeben.

F.2 `connected`-Befehl

Der `connected`-Befehl gibt auf die Konsole aus, ob der Startflughafen v_s direkt (ohne Zwischenlandung) mit dem Zielflughafen v_t verbunden ist.

Eingabeformat `connected <Startflughafen>;<Zielflughafen>`

`<Startflughafen>` und `<Zielflughafen>` sind jeweils durch ihren IATA-Code repräsentiert wie in Abschnitt B beschrieben.

Ausgabeformat Falls $e \in E$, wird `1` ausgegeben. Falls `<Startflughafen>` und `<Zielflughafen>` gültige Knoten sind, aber $e \notin E$, wird `0` ausgegeben. Im Fehlerfall wird eine aussagekräftige Fehlermeldung ausgegeben.

F.3 `distance`-Befehl

Der `distance`-Befehl gibt die Flugdistanz $distance(e)$ zwischen einem Startflughafen v_s und einem Zielflughafen v_t auf die Konsole aus, falls die Kante $e = (v_s, v_t)$ existiert.

Eingabeformat `distance <Startflughafen>;<Zielflughafen>`

`<Startflughafen>` und `<Zielflughafen>` sind jeweils durch ihren IATA-Code repräsentiert wie in Abschnitt B beschrieben.

Ausgabeformat Falls $e \in E$, wird die Flugdistanz $distance(e)$ ausgegeben. Falls `<Startflughafen>` und `<Zielflughafen>` gültige Knoten sind, aber $e \notin E$, wird `inf` (für *unendlich*) ausgegeben. Im Fehlerfall wird eine aussagekräftige Fehlermeldung ausgegeben.

F.4 `routes`-Befehl

Der `routes`-Befehl gibt auf die Konsole aus, wie viele Routen von einem bestimmten Startflughafen zu einem bestimmten Zielflughafen verlaufen, die genau z Zwischenlandungen enthalten.

Dazu wird die Adjazenzmatrix A gebildet, anschließend die Matrix A^{z+1} berechnet und aus letzterer Matrix die angefragte Routenanzahl abgelesen.

Eingabeformat `routes <Startflughafen>;<Zielflughafen>;<Zwischenlandungen>`

`<Startflughafen>` und `<Zielflughafen>` sind jeweils durch ihren IATA-Code repräsentiert wie in Abschnitt B beschrieben. `<Zwischenlandungen>` ist eine ganze Zahl zwischen 0 und $|V| - 2$.

Ausgabeformat Im Erfolgsfall wird die berechnete Routenanzahl ausgegeben. Im Fehlerfall wird eine aussagekräftige Fehlermeldung ausgegeben.

F.5 `shortest`-Befehl

Der `shortest`-Befehl gibt die Distanz des kürzesten Pfades zwischen einem bestimmten Startflughafen und einem bestimmten Zielflughafen auf die Konsole aus.

Dazu wird die Distanzmatrix D gebildet, anschließend mit Hilfe des Floyd-Algorithmus die kürzesten Distanzen berechnet und aus der Ergebnismatrix die kürzeste Distanz abgelesen.

Eingabeformat `shortest <Startflughafen>;<Zielflughafen>`

`<Startflughafen>` und `<Zielflughafen>` sind jeweils durch ihren IATA-Code repräsentiert wie in Abschnitt B beschrieben.

Ausgabeformat Im Erfolgsfall wird die berechnete Distanz ausgegeben. Falls kein Pfad zwischen `<Startflughafen>` und `<Zielflughafen>` existiert, wird `inf` (für *unendlich*) ausgegeben. Im Fehlerfall wird eine aussagekräftige Fehlermeldung ausgegeben.

F.6 `quit`-Befehl

Dieser Befehl beendet das Programm. Dabei findet keine Konsolenausgabe statt.

F.7 Beispielinteraktion

Beachten Sie im Folgenden, dass Eingabezeilen mit dem `>` -Zeichen eingeleitet werden, gefolgt von einem Leerzeichen. Diese beiden Zeichen sind ausdrücklich kein Bestandteil des eingegebenen Befehls, sondern dienen nur der Unterscheidung zwischen Ein- und Ausgabe.

Beispielinteraktion für den Flugrouten-Graphen aus Abbildung 1

```
> airport FKB
Karlsruhe_Baden-Baden
> connected FKB;HAM
1
> distance FKB;HAM
556
> distance FKB;FRA
inf
> shortest FKB;FRA
967
> routes FKB;FRA;1
2
> quit
```

Anhang: Zeilenweises Lesen einer Datei

Sie dürfen folgendes Code-Beispiel benutzen, um in einer Schleife die Zeilen aus einer gegebenen Datei auszulesen. Für die Fehlermeldungen wurden Konstanten benutzt, die Sie definieren und anpassen müssen. Das `TODO` markiert die Stelle, an der Sie Ihren Code zum Verarbeiten der Zeile einhängen können. Schauen Sie zur Verarbeitung der Zeile die API-Dokumentation der `String`-Klasse[2] an. Beachten Sie insbesondere die `split`-Methode. Die für die Eingabe verwendeten Klassen finden Sie im Paket `java.io`[3].

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println(USAGE);
        System.exit(1);
    }

    FileReader in = null;

    try {
        in = new FileReader(args[0]);
    } catch (FileNotFoundException e) {
        System.out.println(ERROR_MESSAGE);
        System.exit(1);
    }

    BufferedReader reader = new BufferedReader(in);
    try {
        String line = reader.readLine();
        while (line != null) {
            // TODO: process line here
            line = reader.readLine();
        }
    } catch (IOException e) {
        System.out.println(ERROR_MESSAGE);
        System.exit(1);
    }
}
```

[2] http://docs.oracle.com/javase/7/docs/api/java/lang/String.html
[3] http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html