Algorithms and Data structures
Algorithmen und Datenstrukturen

14. Juni 2018

Praktikum 5 - Max Flow

Version 1.0 - deutsch

Submission Deadline: 1. Juli 2018 @ 23:59

Submission System:
`https://aud.cdc.informatik.tu-darmstadt.de/cgi-bin/index.cgi`

1 Einleitung

Die Stadt Iksburg ist eine kleine, alte Stadt an einer großen Autobahn. Ein ungünstiger Ort, da die Autobahn für Reparaturen vorübergehend gesperrt werden und ein Teil des Verkehrs durch die engen Straßen von Iksburg umgeleitet werden muss. Das Unternehmen welches für die Instandsetzung der Autobahn zuständig ist, muss herausfinden, wie viele Autos sie pro Stunde durch Iksburg umleiten können. Für eine Schätzung, nutzen sie eine Straßenkarte der Altstadt.

Die Straßenkarte ist durch einen gerichteten Graphen dargestellt. Dabei sind Knoten Kreuzungen und Kanten die Straßen. Jeder Straße ist eine Nummer zugeordnet, welche die Anzahl der Autos pro Stunde angibt, die in einer Stunde auf dieser Straße fahren können. Wir gehen davon aus, dass die Kreuzungen beliebig viele Autos aufnehmen können.

Der Projektleiter (der keinen Informatikunterricht hatte) behauptet, die Karte sei nutzlos und will die ankommenden Autos einfach in die Stadt umleiten, um herauszufinden, wie viele Autos in einer Stunde durchfahren werden. Sie, als junge/r Informatikstudierende/r, sind der Meinung, dass ein Programm welches den maximalen Fluss von Autos auf Basis einer vorgegebene Karte, der elegantere Weg ist.

2 Aufgabenstellung

Ihre Aufgabe ist es, eine Java-Klasse `MaxFlow` zu implementieren, die den maximalen Durchfluss von Autos (in einer Stunde) berechnet, die zwischen einer bestimmten Anzahl von Startpunkten (Quellen) und einer bestimmten Anzahl von Endpunkten (Zielen) fahren können. Der Konstruktor der Klasse bekommt als Eingabe den Namen einer Datei, welche die Eingabekarte enthält. Die Klasse muss Methoden zur Berechnung des maximalen Durchflusses in Bezug auf die Anzahl der Fahrzeuge bereitstellen und die Ausgabe entweder als ganze Zahl (z.B. 25 Fahrzeuge/Stunde) oder als neue Karte, in der die Straßen mit ungenutzter Kapazität hervorgehoben werden, angeben.

Die Eingabekarte wird als gerichteter Graph mit Knoten und Kanten dargestellt. Jeder Knoten steht für eine Kreuzung und jede Kante für eine Straße. Jedem Knoten ist ein Name zugeordnet. Jeder Kante ist ein Wert zugeordnet, der die maximale Anzahl von Autos angibt, die die Straße in einer Stunde passieren können. Ihre Klasse soll eine neue Karte ausgeben, in der die Straßen, deren maximale Kapazität nicht erreicht wird, durch fettgedruckte (grundsätzlich dickere) Kanten gekennzeichnet werden. Auch in diesem Praktikum wird die `dot` Sprache für die Darstellung der Karte verwendet, sowohl für die Eingabe als auch für die Ausgabe. Sie dürfen Code aus früheren Praktika verwenden, solange es sich um Ihren eigenen Code handelt. Wir verweisen auf Abschnitt 5 für Beispiele in denen die Kartendarstellung gezeigt wird.

Benutzen Sie den **Ford-Fulkerson** Algorithmus um die maximale Anzahl von Autos zu finden, die die Route von A nach B in einer Stunde zurücklegen können.

3 Der Code

Wir stellen die folgenden Java-Klassen und Tests in dem Zip-Archiv **Lab5.zip** zur Verfügung, welches das Java-Projekt und seine Bausteine enthällt. Die Klassen sind in zwei "packages" aufgeteilt – das "frame" und das "lab" package. Bitte implementieren Sie Ihre Lösung basierend auf dem im Zip-Archiv zur Verfügung gestellten Codegerüst.

3.1 Das 'frame' package

Das Abgabesystem nutzt eigene Kopien dieser Klassen wenn Ihr Quellcode auf dem Server getestet wird. Mögliche lokale Änderungen die Sie an diesem package vornehmen, können vom Abgabesystem also nicht berücksichtigt werden.

3.1.1 `AllTests.java`

Dies ist die JUnit Test-Klasse. Die Testfälle werden benutzt um die Korrektheit Ihrer Lösung zu überprüfen.

3.2 Das 'lab' package

Dieses package enthällt die Dateien, welche Sie modifizieren dürfen. Sie dürfen neue Klassen (in neuen Dateien innerhalb des packages, aber bitte nicht in subpackages), als auch neue Methoden oder Variablen hinzufügen, solange **Sie die Signatur (*Name, Parameter Typ und Anzahl*) der gegebenen Methoden nicht ändern**, da diese von den JUnit Tests verwendet werden. Änderungen außerhalb des "lab" package werden ignoriert, wenn Sie Ihren Quellcode über das Abgabesystem hochladen.

3.2.1 `MaxFlow.java`

Für die Klasse `MaxFlow` sind folgende Methoden zu implementieren:

- `public MaxFlow(String filename)`

  Der Konstruktor der Klasse bekommt als Eingabe den Namen der Datei, die die Karte enthält. Sie können davon ausgehen, dass sich die Datei im gleichen Verzeichnis wie das ausführbare Programm befindet.

- `public int findMaxFlow(String[] sources, String[] destinations)`

  Diese Methode gibt die maximale Anzahl von Autos zurück, die den Weg zwischen den durch das Array `sources` angegebenen Kreuzungen und den durch das Array `destinations` angegebenen Kreuzungen zurücklegen können. Der erste Parameter, `sources`, ist ein Array, das die Namen der Quellkreuzungen enthält und `destinations` stellt die Kreuzungen dar, wo die Stadt endet. Stellen Sie sich zum Beispiel die Situation vor, in der Iksburg, eine Zitadellenstadt, 3 Einfahrtstore und 5 Ausfahrtstore hat. Der maximale Fahrzeugfluss wird für die auf der Karte vorhandenen Teilmengen von Quellen und Zielen berechnet. Wenn kein Name aus dem

Array `sources` auf der Karte ist, wird -1 zurückgegeben, wenn kein Name aus dem Array `destinations` auf der Karte ist, wird -2 zurückgegeben, wenn aus beiden Arrays kein Name vorhanden ist wird -3 zurückgegeben. Wenn kein Pfad gefunden wird, wird -4 zurückgegeben. Wenn `sources` identisch mit `destinations` ist, sollte `Integer.MAX_VALUE` zurückgegeben werden.

- `public ArrayList<String> findResidualNetwork (String[] sources, String[] destinations)`

  Diese Methode gibt die Straßen zurück, auf denen die Kapazität nicht vollständig genutzt wurde, zwischen den Quellen und den Zielen, die durch das Array `sources` bzw. das Array `destinations` angegeben wurden. Es wird eine `ArrayList<String>` zurückgegeben, die die Ausgabekarte enthält, wobei jede Kante, die eine Straße darstellt, deren Kapazität nicht vollständig genutzt wurde, fett markiert werden sollte, wie in Abschnitt 4 beschrieben. Die Beschriftungen aller Kanten werden entsprechend der verwendeten Kapazität geändert und Quell- und Zielknoten entsprechend markiert (siehe Abschnitt 4). Jedes Element in der `ArrayList` entspricht einer Zeile der Ausgabekarte. Die Ausgabe sollte direkt interpretierbarer `dot` Code sein. Die Kreuzungen werden durch ihre Namen identifiziert, wie sie in der Eingabedatei angegeben sind.

  Wenn sich keine der Quellen oder Ziele auf der Karte befinden oder kein Pfad gefunden wird, sollte die zurückgegebene Karte die ursprüngliche Karte ohne markierte Ränder sein. In jedem Fall soll der Inhalt der Eingabedatei nicht verändert werden.

3.3 Test-Dateien

Wir stellen Ihnen vier Eingabedateien zum Testen zur Verfügung:

- `Iksburg1`
- `Iksburg2`
- `Iksburg3`
- `Iksburg4`

Wir empfehlen Ihnen, Ihre Lösung auch mit zusätzlichen Eingabedateien zu testen. Beachten Sie, dass Sie Ihre eigenen JUnit-Testfälle schreiben müssen, um mit Ihren benutzerdefinierten Eingabegraphen arbeiten zu können. Denken Sie über die Annahmen nach, die bei der Eingabe gemacht wurden. Um sicherzustellen, dass Ihre Lösung funktioniert, testen Sie sie mit allen bereitgestellten Testfällen. Neben den angegebenen Eingabedateien testet das Einreichsystem Ihre Lösung mit mehreren zusätzlichen Eingabedateien, um die Richtigkeit Ihres Programms zu bestätigen. Diese zusätzlichen Eingabedateien werden Ihnen zur Verfügung gestellt, sind aber nicht in der Zip-Datei enthalten.

3.4 Zusätzliche Hinweise

- Code aus früheren Praktika darf wiederverwendet werden, sofern es sich um Ihren eigenen Code handelt.
- Beginnen Sie mit der Implementierung einer Klasse zur Darstellung von Kanten und einer Klasse zur Darstellung von Knoten, um den Graphen schließlich als Liste von Kanten und eine Liste von Knoten darzustellen.
- Verwenden Sie den Konstruktor der Klasse `MaxFlow`, um die Testdatei in Ihre Diagrammdarstellung einzulesen.

4 Kanten, Quellen und Ziele in `dot`

Als Ein- und Ausgabeformat für dieses Praktikum werden wir eine Untermenge der `dot` Sprache für gerichtete Graphen verwenden. Dieser Abschnitt beschreibt nur diese Untermenge. Weitere Informationen und Download-Tools finden Sie in der Online-Dokumentation:

`https://www.graphviz.org`

Angenommen, wir haben in der Eingangskarte der Stadt die folgende Zeile, die eine Kante beschreibt deren Kapazität 10 Autos pro Stunde ist:[1].

```
A -> B [label="10"];
```

Im Graphen werden alle Quellen durch fett gedruckte Doppelkreise und alle Ziele durch fett gedruckte Kreise dargestellt. Das entsprechende markup in dot Code wird weiter unten beschrieben. Wenn A die Quelle und B das Ziel ist dann, je nachdem wie viel von der Kapazität dieser Straße tatsächlich genutzt wurde, wird die Kante durch eine fette Linie dargestellt wenn noch etwas Kapazität vorhanden ist, und mit einer normalen Linie wenn 100% verwendet wurden. Die Bezeichnung der Kante ändert sich von der reinen Anzeige der ursprünglichen Kapazität in folgendes Format `<original capacity> - <used capacity>`. Zum Beispiel:

1. bei voller Auslastung:

   ```
   A -> B [label="10-10"];
   ```

2. falls nur ein Teil verwendet wurde (z.B. 8 von maximal 10 Fahrzeugen):

   ```
   A -> B [label="10-8"][style=bold];
   ```

[1] Da wir Autos zählen, können Sie davon ausgehen, dass alle Zahlen ganze Zahlen sind

Die Markierung der Quellen und Ziele erfolgt wie folgt. Wenn es mehrere Quellen und mehrere Ziele gibt, sollten alle markiert werden:

```
A [shape=doublecircle][style=bold];
B [shape=circle][style=bold];
```

Für weitere Beispiele verweisen wir auf Abschnitt 5. Die Namen der Knoten können auch tatsächliche Namen von Orten sein, die Klein- und Großbuchstaben und "'_" enthalten. Zum Beispiel "Taunus_Platz". Innerhalb von Namen sollen keine Leerzeichen verwendet werden.

5 Examples

Abbildung 2 stellt ein Beispiel für eine Eingabekarte dar.

```
digraph {
A -> B [label="10"];
A -> C [label="10"];
A -> D [label="8"];
B -> C [label="3"];
B -> E [label="5"];
C -> E [label="15"];
D -> E [label="3"];
}
```

Abbildung 1: Die Eingabedatei

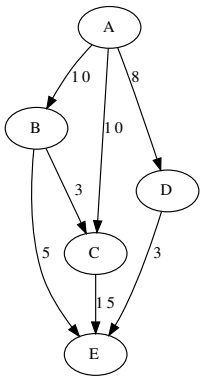

Abbildung 2: Der Graph, der durch den `dot` Code in Abbildung 1 beschrieben wird.

Der Aufruf der Methode `findMaxFlow ("A", "E")` für diesen Graph sollte `21` zurückgeben. Der Aufruf von `findResidualNetwork ("A", "E")` gibt ein `ArrayList` Objekt zurück, das als Elemente die Zeilen des `dot` Codes in Abbildung 3 enthält.

```
digraph {
A -> B [label="10-8"][style=bold];
A -> C [label="10-10"];
A -> D [label="8-3"][style=bold];
B -> C [label="3-3"];
B -> E [label="5-5"];
C -> E [label="15-13"][style=bold];
D -> E [label="3-3"];
A [shape=doublecircle][style=bold];
E [shape=circle][style=bold];
}
```

Abbildung 3: Der Code für Abbildung 4

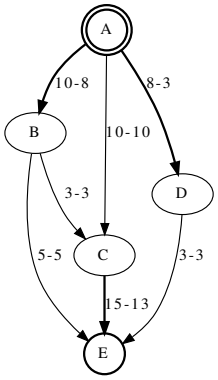

Abbildung 4: Der Graph ist das Ergebnis nach dem Aufruf von `findResidualNetwork ("A", "E")`

Der Aufruf von `findResidualNetwork (sources, destinations)` (mit `sources = "A", "B"` und `destinations = "D", "E"`) gibt ein `ArrayList` Objekt zurück, das als Elemente die Zeilen des `dot` Codes in Abbildung 5 enthält. In diesem Fall ist die Lösung nicht eindeutig, abhängig vom Algorithmus zur Ermittlung des Augmentationspfades.*

```
digraph {
A -> B [label="10-0"][style=bold];
A -> C [label="10-10"];
A -> D [label="8-8"];
B -> C [label="3-3"];
B -> E [label="5-5"];
C -> E [label="15-13"][style=bold];
D -> E [label="3-0"][style=bold];
A [shape=doublecircle][style=bold];
B [shape=doublecircle][style=bold];
D [shape=circle][style=bold];
E [shape=circle][style=bold];
}
```

Abbildung 5: Der Code zur Abbildung 6

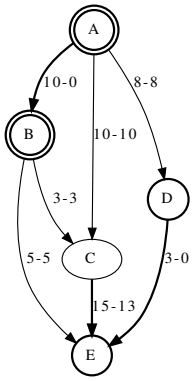

Abbildung 6: Der Graph ist das Ergebnis nach dem Aufruf von `findResidualNetwork (("A", "B"), ("D", "E")`

* Tatsächlich kann die Kante von A nach B "10-8" sein, je nachdem, wie der Augmentationsweg gewählt wurde.