Algorithms and Data structures
Algorithmen und Datenstrukturen

1. Juni 2018

Praktikum 4 - Algorithmus von Dijkstra

Version 1.0 - deutsch

Submission Deadline: 17. Juni 2018 @ 23:59

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

1 Einleitung

GPS-basierte Navigationssysteme ermöglichen es, die kürzeste Route zwischen zwei Punkten, A und B, zu finden. In der Regel können Sie Routenoptionen festlegen, wie z.B. die schnellste oder die kürzeste Route (welche möglicherweise nicht gleiche sind). Aus dem Wissen um die Entfernungen und die Geschwindigkeitsbegrenzungen auf den Straßen berechnet das System eine geschätzte Ankunftszeit am Zielort. Ein Problem bei dieser Art von Berechnungen ist, dass die Zeit, die benötigt wird, um von Punkt A nach B zu gelangen, in Wirklichkeit nicht nur von den Geschwindigkeitsbegrenzungen und der Entfernung abhängt, sondern auch von der Anzahl der Kreuzungen, roter Ampeln und andere Faktoren. In diesem Praktikum geht es darum, einen einfachen Navigationsalgorithmus zu schreiben, der diese Bedingungen berücksichtigt.

2 Aufgabenstellung

Ihre Aufgabe ist es, eine Java-Klasse (`Navigation`) zu implementieren, die den kürzesten Weg zwischen zwei Punkten auf einer Karte berechnet. Die Klasse bekommt als Eingabe den Namen einer Datei, welche die Eingabekarte enthält. Die Klasse soll Methoden zur Berechnung des kürzesten Weges in Bezug auf Entfernung oder Zeit zur Verfügung stellen. Die Ausgabe entspricht entweder einer ganzen Zahl (z.B. 25 km oder 56 Minuten) oder einer neuen Karte, auf der die kürzeste (oder schnellste) Route angezeigt wird.

Die Eingabekarte wird als gerichteter Graph mit Knoten und Kanten dargestellt. Jeder Knoten stellt eine Position dar (beispielsweise eine Kreuzung), und jede Kante stellt eine Straße zwischen den Positionen dar. Jeder Knoten hat einen Namen und eine Zeit (in Minuten) welcher der durchschnittliche Wartezeit an dieser Kreuzung entspricht. Jede Kante hat eine zugehörige Geschwindigkeitsbegrenzung (in km/h) und eine Distanz (in km). Ihre Klasse soll eine neue Karte ausgeben, in der die kürzeste (oder schnellste) Route markiert ist. Die Route wird dadurch markiert, dass die entsprechenden Kanten der kürzesten (oder schnellsten) Route fett (also dicker) darstellt werden. Für dieses Praktikum ist die `dot` Sprache für die Darstellung der Karte, sowohl für die Ein- als auch für die Ausgabe, zu verwendet. Siehe Abschnitt 4 für weitere Informationen zur Darstellung der Karten.

Sie sollen den **Dijkstra-Algorithmus** verwenden, um die kürzeste (oder schnellste) Route von A nach B zu finden. Ihre Lösung muss nicht die kürzesten Routen zu allen Orten von A berechnen, sondern nur die eine nach B. Wenn Sie nach der schnellsten Route gefragt werden, können Sie davon ausgehen, dass das Auto die Höchstgeschwindigkeit für die gesamte Strecke halten kann. Außerdem wird die Wartezeit am Start- und Zielort ignoriert und es wird nur die Wartezeit an Orten zwischen Start- und Zielort berücksichtigt. Einige Knoten können eine Wartezeit von 0 Minuten haben, was bedeutet, dass es sich nicht um Kreuzungen, sondern um andere Orte handelt. Verwenden Sie bei der Berechnung der benötigten Distanz (oder Zeit) den eingebauten Java-Typ `double`. Wenn Sie eine Antwort zurückgeben, geben Sie sie als ganze Zahl (`int`) zurück. Dabie sollen Kommazahlen nach nach oben aufgerundet werden (also werden sowohl 5,25

als auch 5,75 zu 6 aufgerundet). Dazu können Sie die Methode `Math.ceil` verwenden.

3 Der Code

Wir stellen die folgenden Java-Klassen und Tests in dem Zip-Archiv **Lab4.zip** zur Verfügung, welches das Java-Projekt und seine Bausteine enthällt. Die Klassen sind in zwei "packages" aufgeteilt – das "frame" und das "lab" package. Zusätzlich stellen wir die korrekten Ausgabekarten zu den Testfällen und den bereitgestellten Testdateien in **Lab4Output.zip** zur Verfügung. 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 `Navigation.java`

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

- `public Navigation(String filename)`

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

- `public ArrayList<String> findShortestRoute(String A, String B)`

  Diese Methode gibt die kürzeste Route (die Route mit der kürzesten Entfernung) von Punkt A nach Punkt B zurück. Sie gibt eine `ArrayList<String>` zurück, die die Ausgabekarte enthält. Jede Kante auf dem kürzesten Weg von A nach B sollte fett markiert sein, wie in Abschnitt 4 beschrieben. Jedes Element in der `ArrayList`

entspricht einer Zeile der Ausgabekarte. Die Ausgabe sollte direkt interpretierbar sein. Für die in Abbildung 6 dargestellte Karte würde beispielsweise die Ausgabe ArrayList die Länge 20 haben. Die beiden Punkte werden durch ihren Namen identifiziert, wie er in der Eingabedatei angegeben ist. Der Inhalt der Eingabedatei soll nicht verändert werden.

Wenn sich ein oder beide Punkte nicht auf der Karte befinden oder wenn es keinen Weg von A nach B gibt, sollte die zurückgegebene Karte die ursprüngliche Karte sein, ohne markierte Kanten.

Die Reihenfolge, in der die Kanten in der Ausgabekarte erscheinen, muss nicht identisch mit der Eingabekarte sein. Alle Kanten müssen jedoch irgendwo in der Ausgabekarte vorkommen.

- `public ArrayList<String> findFastestRoute(String A, String B)`

  Diese Methode funktioniert genau wie `findShortestRoute`, findet aber stattdessen die *schnellste* Route. Die Ausgabe soll wieder die Eingangskarte sein, mit fett markierten Kanten auf der schnellsten Route.

- `public int findShortestDistance(String A, String B)`

  Diese Methode liefert die kürzeste Entfernung in Kilometern zwischen den Punkten A und B. Wenn Punkt A nicht auf der Karte ist, gibt sie -1 zurück, wenn Punkt B nicht auf der Karte ist, gibt sie -2 zurück, wenn beide nicht auf der Karte sind, gibt sie -3 zurück, wenn es keinen Pfad zwischen Punkt A und Punkt B gibt, gibt sie -4 zurück, wenn Punkt A identisch mit Punkt B ist, sollte sie 0 zurückgeben.

- `public int findFastestTime(String A, String B)`

  Diese Methode liefert die kürzeste Zeit in Minuten zwischen den Punkten A und B. Wenn Punkt A nicht auf der Karte ist, gibt sie -1 zurück, wenn Punkt B nicht auf der Karte ist, gibt sie -2 zurück, wenn beide nicht auf der Karte sind, gibt sie -3 zurück, wenn es keinen Pfad zwischen Punkt A und Punkt B gibt, gibt sie -4 zurück, wenn Punkt A identisch mit Punkt B ist, sollte sie 0 zurückgeben.

3.3 Test Dateien

Wir stellen Ihnen zwei Eingabedateien zum Testen zur Verfügung:

- `Testfile1` und
- `Testfile2`.

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 Abgabesystem Ihre Lösung mit mehreren zusätzlichen

Eingabedateien, um die Korrektheit Ihres Programms zu bestätigen. Diese zusätzlichen Server-Tests werden Ihnen auch zur Verfügung gestellt, sie befinden sich jedoch nicht in dem zip-file.

3.4 Zusätzliche Hinweise

- Beginnen Sie mit der Implementierung einer Klasse zur Darstellung von Kanten und einer Klasse zur Darstellung von Knoten, um den Graphen schließlich als Listen von Kanten und Knoten darzustellen.
- Verwenden Sie den Konstruktor der Klasse `Navigation`, um die Testdatei in Ihre Graphendarstellung einzulesen.

4 Ein- und Ausgabeformat: dot

Als Ein- und Ausgabeformat für dieses Praktikum werden wir eine Untermenge der Sprache `dot` 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`

4.1 dot Sprache

Jede Datei beginnt mit einer Zeile mit dem Wort `digraph`, was bedeutet, dass wir mit gerichteten Graphen arbeiten. Als nächstes folgt ein "{". Die Datei endet mit dem entsprechenden "}". Knoten werden implizit definiert, indem die Kanten zwischen ihnen definiert werden. Beispielsweise sind die beiden Knoten A und B durch eine Kante verbunden. Die Definition eines einfachen Graphen und der entsprechnde Quellcode ist in Abbildung 2 gegeben.

```
digraph {
A -> B;
}
```

Abbildung 1: Code des Graphen

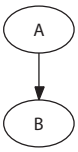

Abbildung 2: Einfacher gerichteter Graph

Man kann die Eigenschaften einer Kante (und natürlich auch von Knoten) ändern, indem man Markup hinzufügt. Wir werden nur zwei Arten von Markup verwenden, nämlich eine Kante dicker zu machen und sie mit einem Etikett zu versehen. Markup wird mit Klammern hinzugefügt wie das Beispiel in Abbildung 3 und 4 zeigt bei dem eine dickerere Kante mit dem Label `10,110` hinzugefügt wurde.

```
digraph {
A -> B [label="10,110"][style=bold];
}
```

Abbildung 3: Code des geänderten Graphens

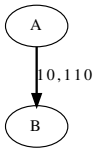

Abbildung 4: Einfacher gerichteter Graph mit dickgezeichneter Kante und Etikett

Wir verwenden Label, um sowohl die Entfernung (in km) zwischen zwei verbundenen Knoten (Länge einer Kante) als auch die Geschwindigkeitsbegrenzung (in km/h) als `"Distance,Speed limit"` darzustellen. Für die Label der Knoten verwenden wir `"Name,Time"`, um den Namen des Knotens und die durchschnittliche Wartezeit (in Minuten) darzustellen, wie im Beispiel in Figure 5 zu sehen ist.

4.2 Beispiel

Die Grafik in Abbildung 6 und der entsprechende Code ist eine der Eingaben für das Praktikum. Wir haben dem Code in Abbildung 5 Zeilennummern hinzugefügt.

Sie können davon ausgehen, dass alle Eingabekarten dem Format in Abbildung 5 folgen, d.h. zuerst erfolden die Kantendefinitionen mit den Beschriftungen und danach dann die Definitionen der Knoten. Der Name eines im Label geschriebenen Knoten ist immer derselbe wie der Knotenname in `dot` (der Knoten `A` in Zeile 2 heißt immer `A` in der Definition des Knoten-Labels, hier in Zeile 12).

4.3 Erstellen von Graphen

`dot` kann verwendet werden, um in viele verschiedene Formate zu exportieren, wie z.B. PostScript, gif, jpeg, png etc. Sie können die neueste Version von der Projektwebseite herunterladen und auf Ihrem Computer ausführen.

```
1.  digraph {
2.  A -> B [label="10,90"];
3.  A -> C [label="8,80"];
4.  C -> B [label="1,50"];
5.  B -> D [label="7,60"];
6.  C -> D [label="6,80"];
7.  D -> E [label="4,90"];
8.  E -> F [label="2,130"];
9.  D -> F [label="5,130"];
10. F -> G [label="5,120"];
11. G -> H [label="5,100"];
12. A [label="A,5"];
13. B [label="B,4"];
14. C [label="C,3"];
15. D [label="D,2"];
16. E [label="E,1"];
17. F [label="F,6"];
18. G [label="G,7"];
19. H [label="H,8"];
20. }
```

Abbildung 5: Beispiel-Code für eine Eingabekarte in dot Sprache

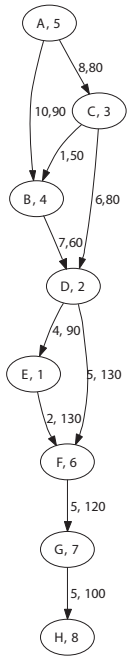

Abbildung 6: Einfacher gerichteter Graph entsprechend dem Code aus Abbildung 5