Algorithmen und Datenstrukturen
Übungsblatt 07

TECHNISCHE UNIVERSITÄT DARMSTADT

Rückfragen zu diesem Übungsblatt vorzugsweise im moodle-Forum zu diesem Blatt!

Sommersemester 2021	v04
Themen:	Algorithmus von Dijkstra
Relevante Foliensätze:	Foliensatz/Video zu Dijkstra
Abgabe der Hausübung:	16.07.2021 bis 23:50 Uhr

Die Aufgabe auf diesem Übungsblatt besteht darin, den Algorithmus von Dijkstra mit einem hohen Allgemeinheitsgrad zu implementieren und auf einen Anwendungsfall anzuwenden, der vielleicht auf den ersten Blick überraschend ist.[1]

Wir werden die Typen für die Knoten (vertices) und die Längen der Kanten (arcs) generisch halten, in Form von Typparametern `V` und `A`. Zudem werden wir die Operationen, die bei Kürzeste-Pfade-Algorithmen wie *Dijkstra* auf den Kantenlängentyp angewandt werden, durch austauschbare Strategien repräsentieren. Hiefür werden Sie die Pfadlängen durch Objekte des Typs `Monoid` aufsummieren und durch Objekte des Typs `java.util.Comparator` miteinander vergleichen. Auf erstere werden wir insbesondere in Teilaufgabe H4 genauer eingehen.

In den meisten Teilaufgaben werden Sie Methoden implementieren, die bereits in Interfaces für Sie definiert sind. Diese verfügen bereits über ausführliche JavaDoc-Kommentare, die Sie als Orientierung nutzen können. Grundsätzlich gilt: wenn eine Methode in speziellen Fällen eine Exception auslösen soll, dann sollte diese auch eine sinnvolle Fehlermeldung enthalten. Wir werden diese aber der Übersichtlichkeit des Übungsblattes zuliebe hier nicht weiter spezifizieren und stattdessen Ihnen die konkreten Formulierungen überlassen.

H1 Graphen als Datenstrukturen — 7 Punkte

Ihre erste Aufgabe ist, die gerichteten Graphen, auf die Sie später den Dijkstra-Algorithmus anwenden werden, als Datenstruktur in Java zu modellieren. Hierfür ist bereits ein Interface `DirectedGraph<V, A>` in der Codevorlage enthalten. Wie oben erwähnt, repräsentieren die Typen `V` und `A` die Knoten und Kantengewichte des Graphen. Zunächst wollen wir diese nicht einschränken. Die einzige Annahme, die wir über `V` treffen, ist, dass dieser Typ eine sinnvolle Implementation der Methoden `equals` und `hashCode` mit sich bringt.

Eine beispielhafte Verwendung des Interfaces `DirectedGraph` können Sie Listing 1 entnehmen. Eine Visualisierung des durch das Codebeispiel erzeugten Graphen können Sie in Abbildung 1 sehen.

Werfen Sie also einen Blick auf das Interface `DirectedGraph<V, A>` im Package `h07.graph`, welches einen endlichen gerichteten Graphen modelliert. Auf die Knotenmenge eines solchen Graphen soll man über die parameterlose Methode `getAllNodes` zugreifen können, welche eine unveränderliche[2] `java.util.Collection` mit allen Knoten als Rückgabe liefert.

Wenn eine Kante von einem Knoten v auf einen Knoten w zeigt, dann nennen wir w ein *Kind* von v. Um alle Kinder eines Knotens abzurufen soll die Methode `getChildrenForNode` verwendet werden. Diese erhält als Parameter einen Knoten des Graphen und liefert eine unveränderliche `java.util.Collection` mit allen Kindern dieses Knotens als Ergebnis. Sollte der übergebene Knoten dabei den Wert `null` haben, dann soll die Methode stattdessen eine `NullPointerException`

[1] Für weitergehend Interessierte: Sie finden im moodle-Kurs der AuD 2021 einen Ausschnitt aus den Folien der Lehrveranstaltung *Algorithmische Modellierung*. Die Vorlesung wird in jedem Sommersemester angeboten. Die Lehrveranstaltung kann aber auch ohne Live-Vorlesung erfolgreich absolviert werden, und zwar jederzeit, nach Ihrer individuellen zeitlichen Einteilung. Mehr dazu im moodle-Kurs „Algorithmische Modellierung im SoSe 2021“. Ebenfalls fachlich passend zu Übungsblatt 07 und genauso flexibel absolvierbar sind die Lehrveranstaltungen *Effiziente Graphenalgorithmen* und *Optimierungsalgorithmen* in jedem Wintersemester.

[2] Eine unveränderliche `Collection` zeichnet sich dadurch aus, dass die Aufrufe von `add`, `remove`, `clear`, usw. eine `UnsupportedOperationException` werfen. Sie können Methoden wie `Set.copyOf` oder `Collections.unmodifiableCollection` verwenden, um eine solche zu erzeugen.

werfen. Falls der übergebene Knoten nicht im Graphen enthalten ist, dann soll die Methode eine `java.util.NoSuchElementExceptio` werfen.

Um auf das Gewicht einer Kante zwischen zwei Knoten zuzugreifen, soll die Methode `getArcWeightBetween` zur Verfügung gestellt werden. Diese erwartet einen Knoten sowie eines seiner Kinder als Parameter und liefert als Ergebnis das Gewicht der Kante, die vom ersten Knoten zum zweiten zeigt. Sollte einer der beiden Knoten `null` sein, dann soll die Methode stattdessen eine `NullPointerException` werfen. Sollte einer der beiden Knoten nicht im Graphen vorhanden sein oder der zweite Knoten gar kein Kind des ersten Knoten sein, dann soll eine `java.util.NoSuchElementException` mit einer passenden Nachricht geworfen werden. Beachten Sie, dass es sich bei einem Graphen `g` dieser Art um einen gerichteten Graphen handelt und daher die Aufrufe `g.getArcWeightBetween(v1, v2)` und `g.getArcWeightBetween(v2, v1)` für zwei Knoten `v1` und `v2` im Allgemeinen *nicht* dasselbe Ergebnis haben.

Zusätzlich stellt das Interface `DirectedGraph` optionale[3] Methoden zum Verändern eines Graphen zur Verfügung. So soll es durch die Methoden `addNode` und `removeNode` möglich sein, dem Graphen Knoten hinzuzufügen bzw. Knoten aus dem Graphen zu entfernen. Zusätzlich können die Methoden `connectNodes` und `disconnectNodes` neue Kanten zwischen zwei Knoten anlegen, bzw. vorhandene Kanten löschen. Sollte eine dieser Methoden von einer Subklasse von `DirectedGraph<V, A>` nicht unterstützt werden, dann soll die Methode in dieser Subklasse so überschrieben werden, dass sie stattdessen eine `UnsupportedOperationException` wirft.

Ansonsten sollen sich diese Methoden folgendermaßen verhalten: die Methoden `addNode` und `removeNode` erhalten als Parameter jeweils den Knoten, der hinzugefügt bzw. gelöscht werden soll. Wenn dieser Knoten `null` ist, soll die entsprechende Methode eine `NullPointerException` werfen. Außerdem soll es nicht möglich sein, einen Knoten zu einem Graphen hinzuzufügen, der dort bereits vorhanden ist. Genauso soll ein Knoten nicht aus einem Graphen gelöscht werden können, in dem er gar nicht existiert. In diesen beiden Fällen soll die entsprechende Methode eine `IllegalArgumentException` (`addNode`) bzw. `java.util.NoSuchElementException` (`removeNode`) werfen. Die Gleichheit zweier Knoten können Sie einfach durch deren `equals`-Methode abfragen. Wenn ein Knoten aus einem Graphen gelöscht wird, dann sollen auch alle ein- und ausgehenden Kanten entfernt werden.

Um eine Kante zu einem Graphen hinzuzufügen, soll die Methode `connectNodes` dienen. Diese erwartet drei Parameter in dieser Reihenfolge: den Startknoten vom Typ `V`, das Kantengewicht vom Typ `A` und den Zielknoten vom Typ `V`. Keiner dieser drei Parameter darf `null` sein, ansonsten soll die Methode eine `NullPointerException` werfen. Außerdem müssen beide Knoten auch tatsächlich im Graphen enthalten sein. Ist dies nicht der Fall, so soll sie eine `java.util.NoSuchElementException` werfen. Wenn bereits eine Kante vom Startknoten zum Zielknoten vorhanden ist, soll eine `IllegalArgumentException geworfen werden.` In jedem weiteren Fall, soll eine neue Kante zum Graphen hinzugefügt werden, deren Gewicht dem Wert entspricht, der der Methode als zweiter Parameter übergeben wurde.

Zuletzt soll die Methode `disconnectNodes` noch die Möglichkeit bieten, vorhandene Kanten zu entfernen. Sie erwartet auch wieder den Start- und Zielknoten als Parameter vom Typen `V`, die beide nicht `null` sein dürfen (ansonsten wird eine `NullPointerException` geworfen). Außerdem wirft die Methode eine `java.util.NoSuchElementException`, wenn einer der beiden Knoten nicht im Graph enthalten ist oder keine Kante vom Start- zum Zielknoten existiert. In jedem weiteren Fall wird die existierende Kante aus dem Graphen gelöscht. Die beiden Knoten bleiben dabei jedoch erhalten.

Ihre Aufgabe ist es nun, eine neue Klasse namens `DirectedGraphImpl` mit der Sichtbarkeit package-private[4] im Package `h07.graph` zu erstellen. Diese soll das Interface `DirectedGraph` implementieren und wie dieses auch zwei Typparameter `V` und `A` deklarieren, die sie an das Interface übergibt.

Implementieren Sie in dieser Klasse nun alle oben genannten Methoden, sodass sie die beschriebenen Spezifikationen erfüllen. Insbesondere soll ein Objekt vom Typ `DirectedGraphImpl` veränderbar sein, d.h. auch die Methoden `addNode`, `removeNode`, `connectNodes` und `disconnectNodes` sollen mit Funktionalität implementiert werden.

Zusätzlich soll die Klasse `DirectedGraphImpl` über einen parameterlosen Konstruktor mit der Sichtbarkeit package-private verfügen, der den Graphen mit einer leeren Knoten- und Kantenmenge initialisiert. Wie Sie bei der Implementierung von `DirectedGraphImpl` konkret vorgehen, ist dabei Ihnen überlassen. Wir empfehlen Ihnen eine ähnliche Vorgehensweise wie bei der aus der Vorlesung bekannten Implementierung der verketteten Liste durch `ListItem`-Objekte. Beispielsweise könnten Sie eine innere Klasse anlegen, die einen Knoten repräsentiert und dessen Wert zusammen mit

[3] In Anlehnung an `java.util.Collection`. Hier sind die Methoden zum Bearbeiten der `Collection` optional, d.h. Implementationen müssen diese nicht zwingend unterstützen und können alternativ eine `UnsupportedOperationException` werfen.

[4] Als *package-private* bezeichnet man Klassen, Methoden oder Attribute, die nur für Klassen im selben Package sichtbar sind. Dieser Zugriffstyp ist der Standard, der aus dem Weglassen des access modifiers resultiert.

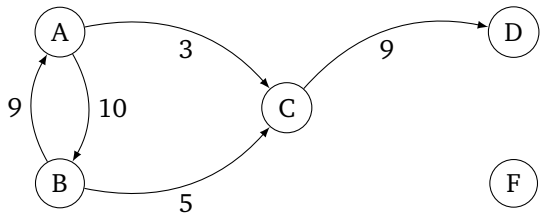

Abbildung 1: Die visuelle Darstellung des Graphen, der vom Code in Listing 1 erzeugt wird.

einer Liste von ausgehenden Kanten speichert. Zusätzlich würden Sie dann eine Klasse für die Kanten anlegen, die das entsprechende Kantengewicht sowie den Zielknoten als Attribut enthält.

Verbindliche Anforderung: Ihre Implementation darf allerdings nicht auf einer Adjanzenzmatrix beruhen (siehe Teilaufgabe H2). Zusätzlich müssen, abgesehen vom Konstruktor und den überschriebenen Methoden, alle weiteren Bestandteile (Methoden, innere Klassen, Attribute, etc.) der Klasse `DirectedGraphImpl` mit der Sichtbarkeit `private` versehen sein.

Listing 1: Ein Beispiel für die Anwendung des Interfaces `DirectedGraph`.

```
DirectedGraph<String, Integer> graph = new DirectedGraphImpl<>();
graph.addNode("A");
graph.addNode("B");
graph.connectNodes("A", 10, "B");
graph.connectNodes("B", 9, "A");
graph.addNode("C");
graph.connectNodes("A", 3, "C");
graph.connectNodes("B", 5, "C");
graph.addNode("D");
graph.addNode("E");
graph.connectNodes("E", 2, "D");
graph.connectNodes("C", 9, "D");
graph.connectNodes("C", 12, "E");
graph.addNode("F");
graph.removeNode("E");
```

H2 Erzeugen von Graphen — 3 Punkte

Bisher ist es noch nicht möglich, Objekte vom Typen `DirectedGraph` außerhalb des Packages zu erzeugen, da Sie die konkrete Implementierung und deren Konstruktor nach außen verborgen haben. Die Motivation hinter diesem Ansatz ist, dass wir die Möglichkeit offen halten wollen, die Implementation auszutauschen oder zu ersetzen. Stattdessen werden Sie sich eine Fabrikklasse vom Typen `DirectedGraphFactory` zunutze machen, um `DirectedGraph`-Objekte zu erzeugen, ohne dabei eine konkrete Implementation auswählen zu müssen.

Das Interface `DirectedGraphFactory` ist bereits im Package `h07.graph` definiert und verfügt über eine parameterlose Methode `createDirectedGraph`, die ein neues Objekt vom Typ `DirectedGraph` erzeugt und als Ergebnis liefert. Wie dieser Graph konkret aufgebaut ist, wird erst durch die Implementation des Interfaces `DirectedGraphFactory` festgelegt.

Eine Möglichkeit, einen endlichen gerichteten Graphen mit n Knoten zu repräsentieren, ist durch eine sogenannte *Adjazenzmatrix*. Hierbei handelt es sich um eine quadratische $n \times n$-Matrix, die die Kantengewichte des Graphen enthält. Der Eintrag in der i-ten Zeile und j-ten Spalte der Matrix entspricht dabei dem Gewicht der Kante vom i-ten Knoten zum j-ten Knoten des Graphen.

Wir wollen die Adjazenzmatrix als Array von Array von `A` namens `adjacencyMatrix` modellieren. Zusätzlich wollen wir die Knoten des zugehörigen Graphen in einem Array von `V` mit dem Namen `nodes` speichern. Hier ist dann der Wert `adjacencyMatrix[i][j]` gleich dem Gewicht der Kante von `nodes[i]` nach `nodes[j]`. Soll zwischen zwei Knoten keine Kante existieren, so ist der entsprechende Eintrag im Array einfach `null`.

Implementieren Sie das Interface `DirectedGraphFactory` nun durch eine `public`-Klasse namens `AdjacencyMatrixFactory` mit denselben generischen Typparametern wie `DirectedGraphFactory`, welche an dieses weitergereicht werden. Die Klasse soll sich auch im Package `h07.graph` befinden. Statten Sie Klasse `AdjacencyMatrixFactory` über die beiden oben genannten Attribute aus, indem Sie diese als private Objektkonstanten anlegen. Der Typ von `nodes` sollte dann natürlich `V[]` sein und der von `adjacencyMatrix` entsprechend `A[][]`.

Erweitern Sie die Klasse `AdjacencyMatrixFactory` zusätzlich um einen öffentlichen Konstruktor, der beide Attribute durch entsprechende Parameter initialisiert. Sie müssen die beiden Arrays hierfür nicht kopieren, sondern es genügt, in den Attributen einen Verweis auf den entsprechenden Parameter zu speichern. Zusätzlich soll der Konstruktor jedoch noch eine `NullPointerException` mit der passenden Nachricht werfen, falls einer der Parameter oder einer der Einträge in `nodes` den Wert `null` hat. Anderenfalls soll auch eine `IllegalArgumentException` geworfen werden, falls die Anzahl der Zeilen oder Spalten in `adjacencyMatrix` nicht der Anzahl der Elemente in `nodes` entspricht.

Implementieren Sie dann die Methode `createDirectedGraph` so, dass sie den entsprechenden Graphen gemäß beider Arrays als neues Objekt vom Typ `DirectedGraphImpl` erzeugt. Der folgende Code-Ausschnitt zeigt, wie sich die Klasse anschließend einsetzen lassen soll, um ein Objekt vom Typ `DirectedGraph` zu erzeugen.

Listing 2: Ein Beispiel für die Anwendung der Klasse `AdjacencyMatrixFactory`, durch die der Graph in Abbildung 1 erzeugt wird.

```
String[] nodes = { "A", "B", "C", "D", "F" };

Integer[][] adjacencyMatrix = {
    { null,   10,    3, null, null },
    {    9, null,    5, null, null },
    { null, null, null,    9, null },
    { null, null, null, null, null },
    { null, null, null, null, null }
};

DirectedGraphFactory<String, Integer> factory = new AdjacencyMatrixFactory<>(nodes, adjacencyMatrix);
DirectedGraph<String, Integer> graph = factory.createDirectedGraph();
```

H3 Pfade als Datenstrukturen — 8 Punkte

Bevor Sie den Algorithmus von Dijkstra implementieren können, fehlt noch eine Möglichkeit, Pfade in Graphen zu speichern und deren Längen zu berechnen.

Hierfür haben wir im Package `h07.graph` das Interface `Path` für Sie definiert. Dieses bietet die Möglichkeit, Pfade zu traversieren und neue Pfade durch das Anhängen einer Kante an einen bestehenden Pfad zu erzeugen.

Ihre Aufgabe ist nun, das Interface `Path` durch eine Klasse `PathImpl` mit Sichtbarkeit package-private zu implementieren. Alternativ können Sie diese Klasse stattdessen auch von `AbstractPath` erben lassen, die `Path` bereits um eine eigene `toString`-Methode erweitert, um Ihnen später das Debugging leichter zu machen. In jedem Fall soll `PathImpl` auch wie `Path` über die beiden Typparameter `V` und `A` verfügen.

Wie Sie bei der Implementation von `PathImpl` genau vorgehen, ist wieder Ihnen überlassen. Wir wollen hier lediglich kurz auf die Methoden und deren Implementationsanforderungen eingehen.

Werfen Sie zunächst einen Blick auf das innere Interface `Path.Traverser`. Dieses bietet ähnlich zu `java.util.Iterator` die Möglichkeit, einen Graphen zu traversieren. Im Gegensatz zu dem Ihnen bereits bekannten Interface bietet es aber auch die Möglichkeit, die Distanz zum nächsten Knoten in Form des Kantengewichtes abzurufen. Ein `Traverser` lässt sich für einen gegebenen Pfad erzeugen, indem dessen parameterlose Methode `traverser` aufgerufen wird. Zunächst soll die Methode `getCurrentNode` dieses `Traverser`-Objektes den Startknoten des zugehörigen Pfades als Ergebnis liefern. Solange noch weitere Knoten im Pfad enthalten sind, soll ein Aufruf von `walkToNextNode` den Pfad weiter entlang laufen und das aktuelle Element auf den Knoten setzen, auf den die ausgehende Kante des vorherigen aktuellen Knoten zeigt. Ob noch weitere Knoten im Pfad enthalten sind, soll sich mit der Methode `hasNextNode` ermitteln lassen. Diese liefert genau dann `true`, wenn es sich beim aktuellen Element nicht um den letzten Knoten des Pfades handelt. Sollte der `Traverser` sich bereits auf dem letzten Knoten befinden, dann soll die Methode `walkToNextNode` eine

`NoSuchElementException` und die Methode `getDistanceToNextNode` eine `IllegalStateException` werfen. Anderenfalls liefert letztere das Kantengewicht der vom aktuell betrachteten Knoten ausgehenden Kante. Eine beispielhafte Anwendung dieses Interfaces können Sie sich in der Klasse `AbstractPath` ansehen.

Zusätzlich zur Methode `traverser`, die einen solchen `Traverser` liefert, soll `PathImpl` auch die Methode `iterator` des Interfaces `Iterable<V>` implementieren. Dieser soll lediglich über die Knoten im Pfad iterieren, unabhängig von den Kantengewichten.

Implementieren Sie schließlich noch die Methode `concat`, die einen weiteren Knoten und eine Distanz als Parameter erwartet und eine Kopie dieses `Path`s liefert, an die der neue Knoten durch eine neue Kante mit der übergebenen Distanz als Gewicht angehängt wurde. Das Objekt vom Typ `Path`, auf das diese Methode aufgerufen wird, soll dabei jedoch nicht verändert werden. Sollte der neue Knoten oder das Gewicht der neuen Kante `null` sein, so soll diese Methode eine `NullPointerException` werfen.

Um einen neuen `Path` zu erzeugen, können Sie die statische Fabrikmethode `of` im Interface `Path` nutzen. Dafür müssen Sie diese zuvor aber noch implementieren, indem Sie die Exception entfernen und durch die auskommentiere Zeile ersetzen.

H4 Operationen auf Kantengewichten — 1 Punkt

Um die Länge eines Pfades für den Dijkstra-Algorithmus zu bestimmen, wird es nötig sein, Kantengewichte aufzusummieren. Bisher haben wir die Typen für Kantengewichte jedoch nicht eingeschränkt, womit es nicht garantiert ist, dass sich diese überhaupt summieren lassen.

Stattdessen wollen wir die notwendige algebraische Struktur als allgemeines Interface `Monoid`[5] definieren und diese das Summieren von Kantengewichten für uns übernehmen lassen.

Hierfür definiert das Interface `Monoid` die Methoden `zero` und `add`. Die Namen beider Methoden sind hier nicht wortwörtlich zu verstehen, da sie beispielsweise auch derart implementiert werden könnten, dass sie die Multiplikation von Zahlen oder die Verkettung von Permutationen modellieren.

Für unsere Beispiele genügt zunächst aber die gewöhnliche Integer-Addition. Erstellen Sie hierfür im Package `h07.algebra` eine neue `public`-Klasse namens `IntegerAddition`, die das Interface `Monoid<Integer>` implementiert. Sie soll lediglich über den parameterlosen Standardkonstruktor verfügen und keinen Zustand haben. Implementieren Sie `add` so, dass sie die Summe der beiden als Parameter übergebenen Ganzzahlen als Ergebnis liefert. Die Methode `zero` soll das neutrale Element der Addition, also die 0, als Ergebnis liefern.

H5 Der Algorithmus von Dijkstra — 7 Punkte

Nun haben Sie alle nötigen Vorbereitungen getroffen, endlich den Algorithmus von Dijkstra zu implementieren. Dieser soll als `public`-Klasse `Dijkstra` mit den Typparametern `V` und `A` im Package `h07.algorithm` angelegt werden und das Interface `ShortestPathsAlgorithm` im selben Package implementieren. Dabei erhält `ShortestPathsAlgorithm` dieselben Typparameter wie `Dijkstra`.

Die wesentliche Methode im Interface `ShortestPathsAlgorithm` ist `shortestPaths`. Diese erwartet als Parameter einen gerichteten Graphen `graph` als Objekt vom Typ `DirectedGraph<V, A>` und einen Startknoten `startNode` vom Typ `V`. Zusätzlich erhält sie noch ein `Monoid<A>` namens `monoid`, mit dem die Kantengewichte addiert werden sollen, sowie einen `java.util.Comparator<? super A>` mit dem Namen `comparator`, der verwendet werden soll, um die Gesamtlänge zweier Pfade miteinander zu vergleichen.

Wichtig ist hier, dass das Monoid derart beschaffen sein soll, dass für alle Kantengewichte `w` des Graphen und alle Pfadlängen `d` gilt: `comparator.compare(monoid.add(d, w), d) >= 0`. Das bedeutet, dass der Graph keine Kanten mit negativen Gewichten (im Sinne des Monoids und der Ordnungsrelation) enthalten darf. Diese Eigenschaft wird vertraglich für jeden Aufruf von `shortestPaths` gefordert, Sie dürfen also davon ausgehen, dass sie immer erfüllt ist.

[5] Einige von Ihnen kennen Monoide vielleicht aus der Veranstaltung *Automaten, formale Sprachen und Entscheidbarkeit*. Für diese Aufgabe müssen Sie jedoch nicht wirklich wissen, worum es sich dabei genau handelt.

Der Rückgabewert von `shortestPaths` soll eine `Map<V, Path<V, A>>` sein, die ein Schlüssel-Wert-Paar für jeden von `startNode` durch einen Pfad erreichbaren Knoten des Graphen beinhaltet. Der Schlüssel soll dafür der Knoten selbst sein, und der entsprechende Wert der kürzeste Pfad, der vom Startknoten dort hin führt. Ein Pfad mit der Länge `d1` ist kürzer als ein Pfad der Länge `d2` wenn `comparator.compare(d1, d2) < 0` gilt. Sollte es für einen Zielknoten mehr als einen kürzesten Pfad geben, so kann die `Map` einen beliebigen dieser Pfade für den entsprechenden Knoten enthalten. Ist ein Knoten nicht vom Startknoten aus erreichbar, dann soll für diesen auch kein Eintrag in der `Map` enthalten sein.

Sollte einer der vier Parameter von `shortestPaths` den Wert `null` haben, dann soll die Methode eine entsprechende `NullPointerException` werfen. Sollte der Startknoten nicht im Graph enthalten sein, dann soll die Methode eine `java.util.NoSuchElementException` werfen.

Das konkrete Vorgehen bei der Implementation der Methode ist wieder Ihnen überlassen. Insbesondere werden Sie sich eine Möglichkeit überlegen müssen, den nächstkürzesten Pfad zu einem noch nicht betrachteten Knoten aus einer Warteschlange ziehen zu können. Hierfür lohnt sich vielleicht ein Blick auf die Klasse `java.util.PriorityQueue`. Bedenken Sie, dass diese jedoch keine Methode anbietet, um den Wert eines Elementes zu verringern. Finden Sie eine alternative Lösung, um dieses Problem zu umgehen?

Warnung: Diese Implementation des Algorithmus von Dijkstra hält nicht die asymptotische Komplexität ein, die in der AuD und in gängigen Quellen für diesen Algorithmus angegeben wird. Der Grund ist, dass die Pfade zu allen Knoten materialisiert sind, so dass schon der dafür benötigte Speicherplatz im Worst Case quadratisch in der Anzahl Knoten ist und daher allein schon die schrittweise Erzeugung der Pfade die normalerweise angegebene asymptotische Komplexität übersteigt. In der Praxis würde man zu jedem Knoten v nicht einen ganzen kürzesten Pfad, sondern nur die letzte Kante eines kürzesten Pfades vom Startknoten zu v speichern und daraus nach Beendigung des Algorithmus von Dijkstra den Pfad rekonstruieren, indem man sich über diese Kanten von v zum Startknoten zurückhangelt. (Vergleiche Zusammenfassung zu Dijkstra im Abschnitt Vorlesung des moodle-Kurses zur AuD.)

Sie finden im Package `h07.experiment` eine Klasse namens `RoadTrip` mit einer `main`-Methode, die ein Beispiel aus der Wikipedia implementiert. Nutzen Sie dieses gern, um ihre Implementierung zu testen!

H6 Das Chicken-Nugget-Problem — 5 Punkte

In dieser Aufgabe werden Sie Ihren Algorithmus nutzen, um ein etwas ungewöhnlicheres Problem[6] zu lösen, bei dem es sich um eine spezielle Form des Frobenius-Problems[7] handelt.

Die Aufgabenstellung ist folgende: eine Fast-Food-Kette bietet ein Produkt in den drei unterschiedlichen Packungsgrößen 6, 9 und 20 an. Die Frage ist nun, wie viele Exemplare jeder dieser Packungen man erwerben muss, um genau eine feste Anzahl der enthaltenen Chicken-Nuggets zu erhalten. Sind wir beispielsweise an genau 38 Chicken-Nuggets interessiert, dann sollten wir eine 20er-Packung und zwei 9er-Packungen erwerben.

Betrachten Sie den Graphen in Abbildung 2. Dieser hat sechs Knoten mit den Werten von 0 bis 5. Aus jedem Knoten gehen zwei Kanten aus: eine mit der Länge 9 (blau) und eine mit der Länge 20 (rot). Sei v der Wert eines Knoten im Graphen, dann ist dieser durch eine blaue Kante mit dem Knoten mit dem Wert $(v + 9) \bmod 6$ verbunden. Ebenso zeigt eine rote Kante von v aus auf den Knoten mit dem Wert $(v + 20) \bmod 6$.

Ihre Aufgabe ist es nun, eine Fabrikklasse für einen solchen Graphen zu erstellen. Legen Sie dafür die `public`-Klasse `ChickenNuggetGraphFactory` im Package `h07.experiment` an, die das Interface `DirectedGraphFactory<Integer, Integer>` implementiert. Überschreiben Sie die Methode `createDirectedGraph` so, dass sie den oben beschriebenen Graph als Rückgabewert liefert. Achten Sie hier insbesondere darauf, dass sich der resultierende Graph nicht verändern lässt. Die Methoden `addNode`, `removeNode`, `connectNodes` und `disconnectNodes` sollten also alle eine `UnsupportedOperationException` werfen. Alle weiteren Methoden sollen sich wie im Interface `DirectedGraph` spezifiziert verhalten.

Sei nun n eine beliebige Anzahl an Chicken-Nuggets. Um eine mögliche Zusammenstellung von Packungsgrößen zu bestimmen, können Sie den folgenden Algorithmus verwenden: zunächst bestimmen Sie den kürzesten Pfad vom Knoten mit dem Wert 0 zum Knoten mit dem Wert $n \bmod 6$. Für jede dabei passierte blaue Kante legen Sie eine Packung mit

[6] `https://youtu.be/gGEwvCLjf8w`
[7] `https://de.wikipedia.org/wiki/Münzproblem`

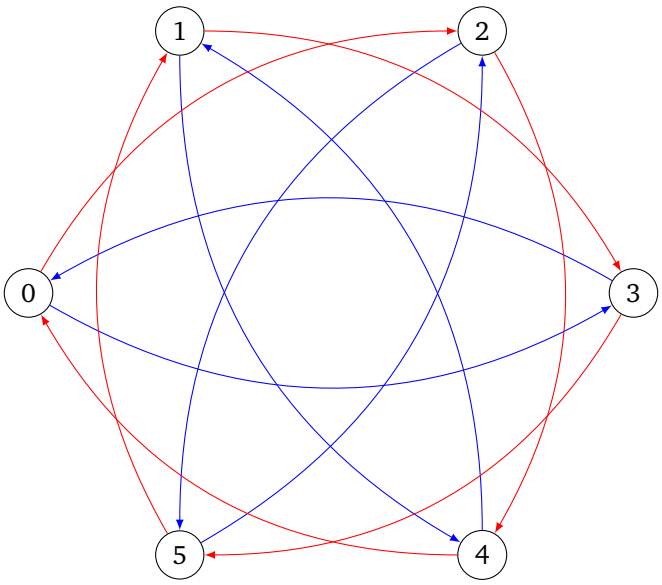

Abbildung 2: Der Chicken-Nugget-Graph. Rot eingezeichnete Kanten haben die Länge 20, blaue die Länge 9.

9 Chicken-Nuggets in den Warenkorb, und für jede rote eine mit 20. Die verbleibende Anzahl sollte sich nun restlos durch 6 teilen lassen. Der entsprechende Quotient ist die Anzahl der Packungen mit 6 Chicken-Nuggets, die sie kaufen müssen.

Setzen Sie diesen Algorithmus in einer `public`-Klasse namens `ChickenNuggets` im Package `h07.experiments` an. Diese soll über eine öffentliche Klassenmethode namens `computePackageNumbers` verfügen, welche einen `int`-Parameter namens `numberOfNuggets` erwartet. Als Ergebnis soll sie ein `int`-Array mit drei Komponenten liefern. In der ersten Komponente soll die Anzahl der benötigten 6er-Packungen stehen, in der zweiten die der 9er-Packungen und in der letzten schließlich die Anzahl der 20er-Packungen. Ein Aufruf von `computePackageNumbers(67)` sollte also das Array `{3, 1, 2}` liefern. Sollte es nicht möglich sein, die geforderte Anzahl exakt mit den angebotenen Packungsgrößen darzustellen, dann soll die Methode `null` als Ergebnis liefern.