DFS und Dijkstra-Algorithmus implementieren

Erweitern Sie den Graphen-Editor mit zwei Algorithmen-Implementierungen zur Bestimmung eines Spannbaums bzw. zur Suche von kürzesten Wegen, um diese Algorithmen genauer und auch auf der Implementierungsebene kennenzulernen.

Wenn Sie bisher den Graphen-Editor nicht installiert haben, oder Information über die Benutzung suchen, schauen Sie bitte auf dem Selbststudiums-Arbeitsblatt zu *Adjazenzlisten* nach[1].

Die zusätzlichen Algorithmen müssen Sie in Ihre Klasse `AdjListGraph` einbauen. Haben Sie diese Klasse nicht selbst vervollständigt, besorgen Sie sich eine vervollständigte Version vom File Server als Grundlage Ihrer Arbeit.

Aufgabe 1: Depth-First-Search

Implementieren Sie den Depth-First-Search Algorithmus in Ihrer Graphenklasse `AdjListGraph`. Damit der Graphen-Editor davon weiss, und den Algorithmus aufrufen kann, muss `AdjListGraph` neu das Interface `GraphAlgorithms.DFS` implementieren. Ergänzen Sie die Klassen-Definition entsprechend.

Die im Interface `GraphAlgorithms.DFS` definierte Methode `Graph<K> traverse(K startVertex)` erhält als Argument einen Startknoten bei dem die Traversierung beginnen soll.

Welche zusätzlichen Attribute werden in der `Vertex`-Klasse benötigt? Seien Sie möglichst zurückhaltend mit neuen Attributen.

Weitere unterstützende Methoden, z.B. für rekursive Abläufe in der Klasse `AdjListGraph` sollten nur `private` Zugriff benötigen.

Aufgaben

a. Implementieren Sie die oben beschriebene Methode `traverse` so, dass jeder erstmals besuchte Knoten auf der Konsole aufgelistet wird (mit `System.out.print(o.data)`). Als Resultat soll der unveränderte Graph zurückgegeben werden (`return this`).

   Zum Ausprobieren Ihres Algorithmus mit dem Graphen-Editor verwenden Sie einen ungewichteten Graphen (z.B. `TSkomplexer.graph`). Markieren Sie genau einen Startknoten (*Mode Selection*). Der Button *DFS* startet dann Ihren Algorithmus.

   Als Ergebnis sollte in der Konsole angegeben werden, welche Knoten in welcher Reihenfolge besucht wurden.[2]

b. Modifizieren Sie die Lösung aus Teil a. so, dass beim Traversieren ein Spannbaum erstellt wird. Dieser soll dann als Resultat zurückgegeben werden. Der Editor stellt ihn dann als Ergebnis der DFS-Operation dar.

   Vorschlag zum Vorgehen:

   i. Die Methode `traverse` erzeugt einen neuen leeren Graphen – den Spannbaum.
   ii. Zusätzlich zur Ausgabe auf der Konsole wird während des Traversierens jeweils ein neuer Knoten im Spannbaum eingefügt (`addVertex` aufrufen, mit dem `data`-Wert des besuchten Knotens!)
   iii. Wird ein neuer (bisher nicht besuchter) Knoten „entdeckt“, wird im Spannbaum eine Kante vom letzten besuchten Knoten zum aktuellen Knoten eingefügt. An `addEdge` sollten wiederum die `data`-Werte der beteiligten Knoten übergeben werden.

[1] “*07-4 Selbststudium Implementierung Adjazenzlisten”*

[2] Bei gerichteten Graphen kann es richtig sein, dass nicht alle Knoten im Graphen besucht werden. Warum nicht?

Aufgabe 2: Kürzester Weg (nach Dijkstra)

Implementieren Sie den Dijkstra-Algorithmus in einer neuen Klasse `Dijkstra` mit einer Methode

```
static <K> Graph<K> getShortestPath(WeightedGraph<K> g, K from, K to)
```

welche im Graphen `g` den kürzesten Pfad zwischen den beiden Knoten `from` und `to` berechnet und als Resultat einen Graphen zurück gibt, der den gefundenen kürzesten Pfad zwischen den beiden Knoten darstellt.

Vorbereitung

Damit der Graphen-Editor diese Methode aufrufen kann, muss die existierende Klasse `WeightedGraphImpl` so verändert werden, dass diese das Interface `GraphAlgorithms.ShortestPath` implementiert. Die dafür nötige Methode `getShortestPath` soll einfach Ihre Implementierung in der Klasse `Dijkstra` aufrufen (eine entsprechend veränderte Version der Klasse `WeightedGraphImpl` liegt auf dem File Server bereit).

Konzept

Zusatzinformationen, die zu einem Knoten abgelegt werden sollen, können in einer `HashMap` gespeichert werden, mit den Knoten als Schlüssel. Diese Idee wird in der Klasse `AdjListGraph` bereits verwendet und auch die Klasse `WeightedGraphImpl` verwendet eine Instanz eines ungewichteten Graphen und speichert die Gewichte in einer separaten Hashtabelle.

Ihre Implementation können Sie im Editor über den *Shortest Path* Knopf aufrufen. Dazu muss im Editor ein gewichteter Graph dargestellt sein, und zwei Knoten müssen darin ausgewählt sein. Als Resultat der Operation wird der von Ihrer Methode `getShortestPath` zurückgegebene Graph dargestellt.

Prüfen Sie Ihren Algorithmus mit dem SBB-Graphen und bestimmen sie die kürzeste Verbindung zwischen Genf und St. Moritz.

Aufgaben:

Wir empfehlen folgendes Vorgehen:

a. Implementieren Sie die Methode `getShortestPath` in der Klasse `Dijkstra` zunächst als $O(|E| + |V|^2)$ Algorithmus. Der Knoten mit der kürzesten Distanz zum Startknoten wird jeweils sequenziell in den *values* der *HashMap* gesucht, oder – effizienter – in einer separaten Tabelle, die nur jene Knoten enthält, für welche die Distanz vom Startknoten noch nicht definitiv bekannt ist, aber bereits ein Weg mit Länge kleiner ∞ gefunden wurde. Für die Implementation einer solchen Tabelle können Sie z.B. die Klasse `java.util.ArrayList` verwenden.

b. Überlegen Sie sich, wie man (mit Hilfe der Informationen in den *via*-Attributen) vom Zielknoten wieder zurück zum Startknoten kommen kann. Implementieren Sie die nötigen Schritte, so dass dabei ein neuer Graph entlang dieses Weges aufgebaut wird. Dieser Graph soll als Ergebnis zurückgegeben werden.

c. (*herausfordernd*) Implementieren Sie den Algorithmus so, dass er einen Laufzeitaufwand von $O((|E| + |V|) \log |V|)$ hat. Das geht, wenn die Knoten mit noch nicht definitiv bekannter Entfernung vom Ausgangsknoten entweder in einem *Heap* oder in einem *Baum* gespeichert werden.

   Falls Sie eine alte *Heap*-Implementation rezyklieren wollen ist dies prima. Falls Ihnen die Anpassungsarbeiten zu gross sind, empfehlen wir Ihnen ein `java.util.TreeSet` oder eine `java.util.PriorityQueue` zu verwenden. `TreeSets` verwenden ausgeglichene Bäume, d.h., der Einfüge- und Löschaufwand ist garantiert $O(\log n)$.

   Durch ersetzen der Tabelle durch einen *Baum* oder einen *Heap* wird die Suche nach dem nächsten zu bearbeitenden Knoten effizienter, aber der Update der Distanzen wird teurer. Diese Updates verändern die Priorität eines Elementes im Heap bzw. den Schlüssel eines Elementes im Baum. Diese Werte dürfen nicht einfach verändert werden, da dadurch die Ordnungsbedingung im Baum oder im Heap verletzt werden können!

   Damit ein `TreeSet` oder eine `PriorityQueue` verwendet werden kann muss die Elementklasse (z.B. `Vertex`) so erweitert werden, dass sie zusätzlich das `Comparable` Interface unterstützt, oder es muss beim Erzeugen des Sets ein Comparator-Objekt übergeben werden z.B. `TreeSet(Comparator c)`, das zwei Knoten vergleichen kann.