Team: 4_2, Marcel Schlegel, Florian Bauer

Aufgabenaufteilung:

i. Aufgaben: GUI, Floyd-Warshall, GraphGenerator
   Dateien: siehe JavaDoc, bzw. https://github.com/flbaue/gka-wise14

ii. Aufgaben: Speichern & Laden, Dijkstra, BFS Iterator, FloydWarshall
   Dateien: siehe JavaDoc, bzw. https://github.com/flbaue/gka-wise14

Quellenangaben: Es wurde kein Quellcode übernommen. Für Algorithmen wurde Wikipedia zur Hilfe genommen.

Bearbeitungszeitraum: Erster GitHub Commit Montag 20.10.14, aktuell letzter GitHub Commit Sonntag, 23.11.14 (https://github.com/flbaue/gka-wise14/commits/master)

Aktueller Stand: Aufgabe 1 & 2 sind Fertig.

Algorithmus:

Allgemein: Wir haben Konten als Vertex Klasse Implementiert, wobei Vertex Instanzen durch den gegebenen Namen eindeutig sind. Jeder Vertex kann ein Marker Objekt enthalten. Marker Objekte dienen dazu während der Verarbeitung durch Algorithmen, Informationen wie Distanz, Vorgänger, Fertig(Visited), etc. zu speichern. Zu beginn eines Algorithmus werden daher alle Marker der im Graph enthaltenen Knoten entfernt. Am Ende kann man so von jedem Knoten die entsprechenden Informationen abfragen.

BFS: Es werden zunächst mögliche Marker aller Knoten entfernt und lediglich der Startknoten bekommt einen Marker mit der Distanz 0 und sich selber als Vorgänger gesetzt. Anschließend wird der Startknoten als erster Knoten in die Queue aufgenommen. Damit ist der Iterator fertig initialisiert und der Nutzer kann beginnen ihn zu benutzen. Dazu wird zunächst die Methode .hasNext aufgerufen. Diese Methode prüft anhand der Queue, ob noch Knoten verfügbar sind und gibt dies zurück. Danach kann mit .next zum nächsten Knoten gesprungen werden. .next holt sich dazu den nächsten Knoten aus der Queue, setzt seinen Marker auf visited, sucht alle noch zu bearbeitenden Nachbarn, und gibt den Koten an den Aufrufer aus. Nachbarknoten werden gefunden in dem über alle direkt verbundenen Knoten iteriert wird und alle Knoten, die noch keinen Marker haben, diesen erhalten und in die Queue eingefügt werden.

Dijkstra: Es werden zunächst alle Knoten vorbereitet, das heißt sie bekommen einen Marker mit „null“ als Vorgänger sowie Integer.MAX_VALUE als Distanz. Der Startknoten erhält sich selbst als Vorgänger sowie die Distanz 0. Anschließend wird in einer Schleife stets der Knoten mit der Geringsten Distanz ermittelt und die Distanzwerte aller seiner nicht fertigen Nachbarn überprüft, und ggf. aktualisiert. Dabei wird auch der Vorgänger entsprechend gesetzt. Kann kein Unbefugter Die Schleife beginnt nun von Vorne. Kann kein nicht fertiger Knoten mehr ermittelt werden, ist der Algorithmus fertig. Detaillierte Ergebnisse, Distanz und Vorgänger, können dann direkt über die Knoten abgefragt werden.

Floyd-Warshal: Es werden zunächst alle Knoten und Kanten des Graphen in geordnete Listen überführt um feste Indizes zu bekommen.
Anschließend wird eine Distanzmatrix initialisiert, wobei alle Felder mit Double.POSITIVE_INFINITY besetzt werden. Ausnahme sind die Felder bei denen Knoten auf sich selber verweisen (Diagonale in der Matrix). Diese Felder werden mit 0 initialisiert.
Als zweites wird eine next-Matrix initialisiert. Diese Matrix dient später der Pfadfindung. Hier werden alle Felder mit -1 initialisiert.
Als nächstes werden nun alle Knoten mit direkten Nachbarn sprich alle Kanten ausgewertet. Das Gewicht der jeweiligen Kante wird in der Distanzmatrix gespeichert. Zusätzlich werden in der next-Matrix für den weg von i nach j steht j als nächster Knoten vermerkt. Gibt es hier mehrere Kanten zwischen den Knoten wird die Kante mit dem geringsten Gewicht verwendet.
Als letztes werden nun die Transitdistanzen ermittelt. Dazu werden die Distanzen von i nach j über k ermittelt und die bestehende i nach j Distanz ggf. aktualisiert. Findet eine Aktualisierung statt, wird auch in der next-Matrix die Referenz für den als nächstes zu besuchenden Knoten auf dem weg von i nach j aktualisiert.

Datenstrukturen & Implementierung

Graph: Wir verwenden für Graphen die Klassen DirectedWeightedPseudograph sowie WeightedPseudograph des JGraphT Frameworks in Kombination mit DefaultWeightedEdge als Kante und der selbst implementierten Vertex Klasse für Knoten.
Der Grund warum wir uns für den Pseudograph entschieden haben ist, da dieser die größte mögliche Freiheit bietet. So sind bei anderen Graph-Klassen Dinge wie doppelte Kanten oder Loops nicht erlaubt.

GraphLoader: Hier wird nur eine einfache Java Hash Map vorgehalten. Sie dient der Speicherung bereits erzeugter Vertex Instanzen und deren einfacher Wiederverwendung unter Zuhilfenahme des String Namens.
Zwischen den Methoden werden Listen von FileEntry Objekten weitergereicht. FileEntry stellt eine Zeile der eingelesenen Datei dar. Die Zeile wird beim erzeugen von FileEntry analysiert und validiert. Dabei werden alle relevanten Informationen extrahiert und sind über die Getter-Methoden des FileEntry Objekts abrufbar. Schlägt die validieren einer Zeile fehl, wird diese Zeile ignoriert.

GraphSaver: Hier werden alle Kanten des Graphen als EdgeSet genommen und jede Kannte in ein FileEntry Objekt überführt. Anschließend werden noch alle Knoten ohne Kanten ermittelt und diese ebenfalls in FileEntry Objekte überführt.
Anschließend werden die String Repräsentationen der FileEntry Objekte zeilenweise in die gegebene Datei geschrieben.

Tests

GraphIO
testReadGraphFromFile: Es wird ein einfacher Testgraph in Form einer Textdatei gelesen und geprüft ob die Menge der der Erzeugten Knoten und Kanten mit den Erwartungen übereinstimmt.
testSaveGraphAsFile: Es wird ein programmatisch erzeugter Graph in eine Datei gespeichert und anschließend geprüft ob diese Datei erwartete Zeilen enthält.
testGraphIOWithGraphGka6: Es wird der gegebene beispiel Graph graph6.gka eingelesen und die Anzahl der erzeugten kanten und Konnten mit den Erwartungen verglichen
testGraphIOWithGraphGka2: Es wird der gegebene beispiel Graph graph2.gka eingelesen und die Anzahl der erzeugten Kanten und Konnten mit den Erwartungen verglichen

BFS
testIterator: Es wird programmatisch ein Graph erstellt und anschließend mittels BFS Iterator mit den Methoden .hasNext und .next iteriert. Dabei wird bei jedem Schritt die Rückgabewerte der Methoden geprüft. Es wird anschließend geprüft ob .hasNext zum Ende False zurück gibt. Zum Schluss wird geprüft ob alle Knoten die erwarteten Vorgänger in Ihren Markern gesetzt bekommen haben.

Dijkstra
testRun: Es wird graph3.gka verwendet und nach Durchführung des Dijkstra geprüft ob zwei gegebene Städte die erwarteten Distanzwerte erhalten haben.
testPrepareVertexes: Es wird geprüft ob nach der Methode prepareVertexes() des Dijkstra wirklich alle Knoten des Graphen einen korrekt gesetzten Marker haben.
debug: Es wird anhand eines Graphen der ursprünglich zufällig erzeugt wurde und zu einem Fehler führte, ob ein erwarteter kürzester Weg korrekt gefunden wird.

FloydWarshall
testFloydWarshall: Es wird ein Graph programmatisch erzeugt und auf diesem der Algorithmus durchgeführt.
testFWDebug: Es wird auf Basis eines ursprünglich erzeugten Graphen der zu fehlen führte geprüft ob ein kürzester Weg gefunden wird.

GraphGenerator
testGenerate: Es wird ein zufälliger Graph erzeugt und geprüft ob dieser die gegebene Menge an Knoten und Kanten hat. Anschließend werden zwei kürzeste Pfade zu diesem Graph hinzugefügt und geprüft ob die Menge an Knoten und Kanten sich entsprechend verändert.

ShortestPath:
test_3_1: Anhand des Graphen graph3.gka wird jeweils mit Dijkstra und FloydWarshall der kürzeste Pfad gesucht und die Ergebnisse verglichen.
test_3_3: Anhand eines zufällige BIG Graphen wird mit Dijkstra und FloydWarshall der kürzeste Pfad gesucht und die Ergebnisse verglichen.

Aufgabe 4

a) Die Algorithmen liefern jeweils den kürzesten Pfad, das heißt das die länge des Pfades bei beiden gleich ist. Ob es tatsächlich der gleiche Pfad ist, ist jedoch nicht garantiert und hängt unter anderen von den Datenstrukturen ab.

b) Wenn es negative Kantengewichte gibt, kann das Ergebnis des Dijkstra von tatsächlich kürzesten Pfad abweichen.

c) Wir bauen einen Psydograph bei dem sowohl Schlingen als auch Mehrfachkalten möglich sind. Da die Kannten steht zufällige Ziele haben, zum Teil auch zufällige Anfängt, können entsprechende Situationen entstehen. Es kann jedoch nicht gesteuert werden.

d) Alle Kannten erhalten ein zufälliges Gewicht zwischen 10 und 100. Wir können einen Pfad gezielt einfügen bei dem jede Kannte ein Gewicht von 1 hat. So können wir einen gewünschten kürzesten Pfad hinzufügen und auf diesen testen.

e) Bei FloydWarshall müsste die datenstruktur next ersetzt werden durch eine Möglichkeit mehrere alternativen zu speichern.
Beim Dijkstra müsste der Marker ebenfalls erweitert werden um mehrere Mögliche Vorgänger mit Ihrer jeweiligen Distanz zu speichern.

f) Es könnte z.B. immer wenn eine Verbindung gefunden wird die die gleiche Distanz hat wie eine bereits gefundene zufällig entschieden werden welcher Pfad gespeichert wird. Eine andere Möglichkeit wäre die Menge der kürzesten Pfade zu ermitteln und zufällig einen dieser Pfade auszugeben.
Des Weiteren könnten noch andere Eigenschaften (außer dem Zufall) mit einbezogen werden wie zum Beispiel eine maximale Pfadauslastung oder eine spezifische Priorität pro Pfad.