Lösungsdokumentation GKAP Aufgabe 1

Team: GKAP04_TeamD, Jannik Sturhann, Mateusz Chylewski

Aufgabenaufteilung: Die Ausarbeitung der Lösung wurde zusammen im AI-Labor vorgenommen. Daher kann nicht gesagt werden, wer genau für welche Aufgabe verantwortlich ist.

Bearbeitungszeitraum:
- 28.03.2018 12:00-18:00 Einarbeitung GraphStream
- 29.03.2018 13:00-20:00 Implementierung laden/speichern
- 04.04.2018 12:00-18:00 Implementierung BFS
- 05.04.2018 13:00-20:00 Bearbeiten der Lösungsdokumentation

Quellenangaben:
Libraries:
Graphstream core - Stellt Graph Datenstruktur bereitstellt, die visuell dargestellt werden kann und auf der gearbeitet werden kann. (github.com/graphstream/gs-core)

GraphStream JavaFX - Bietet Schnittstelle zwischen GraphStream und JavaFx Oberflächen (github.com/graphstream/gs-ui-javafx)

Andere:
gs-app-click - GraphStream JavaFx Beispiel Applikation. CustomMouseManager für Viewer übernommen(github.com/graphstream/gs-app-click)

Grundbegriffe der Graphentheorie - Graphentheoretische Konzepte und Algorithmen, Thema 01, Julia Padberg

Inhaltsverzeichnis

Datenstruktur

In den Aufgaben wird mit einem Graphen gearbeitet. Dieser Graph kann gerichtet sein und kann Multikanten, sowie parallele Kanten haben. Ein Graph besteht aus einem Menge Knoten und aus einer Menge Kanten. Für die Kanten kann angegeben werden, ob sie gerichtet sind. Sie sind standardmäßig nicht gerichtet. Ein Graph wird dann als gerichtet angesehen, wenn für alle Kanten aus der Menge der Kanten das Graphen gilt, dass sie gerichtet sind.

Nachfolgend eine Auflistung der Elemente des Graphen und deren Attribute. Optionale Attribute (können null sein) werden kursiv gekennzeichnet.

Ein Knoten hat folgende Attribute:

Name(String) , *Attribut(Double)*

Eine Kante hat folgende Attribute:

Source(Knoten), Target(Knoten), Gerichtet(Boolean), *Name(String) , Gewicht(Double)*

Algorithmen

Gegeben sei ein Graph G mit zwei ausgezeichneten Knoten s und t.

Es wird gefordert, den kürzesten Weg von s nach t zu finden. Um das zu erreichen verwenden wir zwei Algorithmen. Zunächst verwenden wir einen Breadth-First Search (BFS) Algorithmus, der ausgehend von s alle Knoten markiert. Diese Markierung zeigt die Anzahl Schritte an, die von s zu einem bestimmten Knoten gegangen werden müssen. Der Algorithmus wird beendet, wenn t markiert wurde. Danach wird der kürzeste Weg von t nach s gefunden, indem die Markierung benutzt wird, die zuvor erzeugt wurde. Zuletzt wird der Weg von t nach s umgedreht (Array reverse) und der Weg von s nach t wird zurückgegeben.

Die Algorithmen sind aus dem Script der Vorlesung entnommen.

Breadth First Search

Schritt 1: Man kennzeichne den Knoten s mit 0 und setze i = 0.

Schritt 2: Man ermittle alle nicht gekennzeichneten Knoten in G, die zu den mit i gekennzeichneten Knoten benachbart und erreichbar sind. Falls es derartige Knoten nicht gibt, ist t nicht mit s über einen Weg verbunden. Falls es derartige Knoten gibt, sind sie mit i + 1 zu kennzeichnen.

Schritt 3: Wenn t gekennzeichnet wurde, folgt Schritt 4, wenn nicht, erhöhe man i um eins und gehe zu Schritt 2.

Schritt 4: Die Länge des kürzesten Weges von s nach t ist i + 1. Der Algorithmus wird beendet.

Kürzester Weg für BFS

Die Kennzeichnung des Knoten a wird dabei mit λ(a) bezeichnet.

Der rückverfolgende Algorithmus erzeugt einen Weg v 0 , v 1 , . . . , v λ(t) , so daß v 0 = s; v λ(t) = t ist.

Schritt 1: Man setze i = λ(t) und ordne v i = t zu.

Schritt 2: Man ermittle einen Knoten u, der zu v i benachbart ist und mit λ(u) = i − 1 gekennzeichnet ist. Man ordne v i−1 = u zu.

Schritt 3: Wenn i = 1 ist, ist der Algorithmus beendet. Wenn nicht, erniedrige man i um eins und gehe zu Schritt 2.

Entwurfsentscheidungen der Implementierung

Benutzung von Graphstream

Wir haben uns entschieden, GraphStream als Library für die Verwaltung der Graphen zu verwenden.

Wir kommen zu diesen Entscheidung aufgrund der folgenden Punkte:

- Streaming Schnittstelle, mit der Änderungen des Graphen, auch asynchron, zwischen verschiedenen Instanzen ausgetauscht werden können.

- Schnittstelle zur Visualisierung des Graphen mit JavaFx

- Elementen des Graphen (Knoten, Kanten, Graph) können mithilfe einer Key-Value-Map eigene Attribute angehängt werden.

- Es werden grundlegende Algorithmen bereitgestellt, wie z.B. Nachbarschaft eines Knotens, Knotengrad, angrenzende Kanten, eingehende/ausgehende Kanten, etc.

- Schnittstelle für Layout der Elemente des Graphen in CSS

- Gute Dokumentation online

Benutzung von JavaFx

Wir haben uns entschlossen, die Bedienung unserer Applikation über JavaFx zu realisieren. Wir kommen zu dieser Entscheidung, zum einen, weil gefordert ist, einen Graph visuell darzustellen. Daher ist es naheliegend auch die Bedienung visuell zu gestalten, wo auch GraphStream eine Schnittstelle zu JavaFx anbietet, was die Implementierung erleichtert.

Abbildung der Datenstruktur in GraphStream

Weil GraphStream eine Datenstruktur verwendet, die der unseren sehr Ähnlich ist, haben wir uns entschieden, unsere Datenstruktur direkt auf die von GraphStream abzubilden. Das heißt beim laden und speichern des Graphen wird mit einer Graph Instanz von GraphStream gearbeitet. Damit ersparen wir uns die Implementierung einer eigenen Datenstruktur und einer Schnittstelle, die unsere Datenstruktur auf die von GraphStream überträgt, wenn wir den Graph darstellen wollen. Außerdem umgehen wir damit sämtliche Synchronisationsprobleme, die auftreten können, wenn Daten redundant in zwei Datenstrukturen verwaltet werden.

GraphStream verwaltet einen Graphen als eine Menge von Knoten und einer Menge von Kanten. Knoten, wie Kanten besitzen eine eindeutige ID. Die eindeutige Id der Knoten, ist der Name der Knoten aus unserer Datenstruktur. Da Namen von Kanten doppelt auftreten können, ist die ID von Kanten immer eine zufällige generierte UUID. Alle anderen Attribute, die die Knoten und Kanten

unserer Datenstruktur besitzten, werden über die Key-Value-Map für Attribute gespeichert, die GraphStream bereitstellt.

Laden und speichern des Graphen

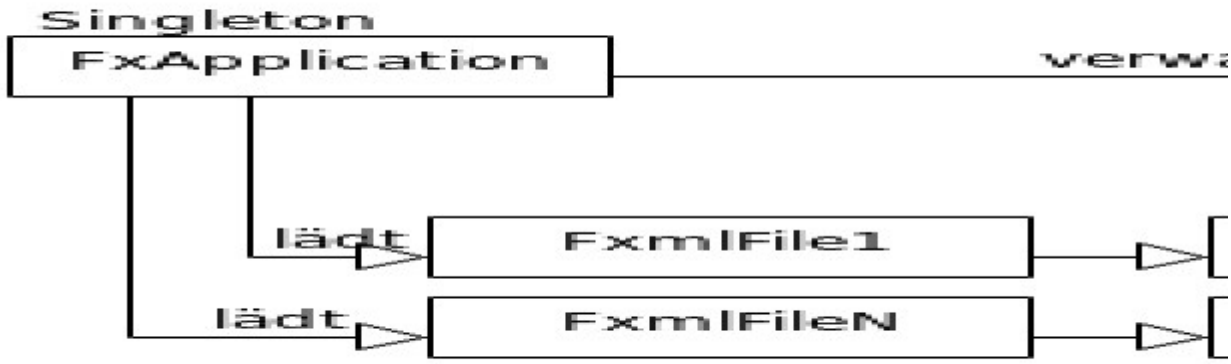
Im Vorheringen haben wir gesagt, dass wir den Graph direkt in die von GraphStream bereitgestellt Datenstruktur importieren und auch wieder aus dieser heraus speichern wollen. Das ergibt das Problem einer Kopplung zwischen der GraphStream API und unserer Implementierung für das lesen und schreiben des vorgegebenen Datenformates.

Deswegen habe wir uns entschlossen, eine Schnittstelle zwischen die Datenverarbeitung und alle Graph Datenstrukturen zu schalten. Somit soll sichergestellt werden, dass wir, wenn nötig, auch andere Implementierungen der Datenstrukturen verwenden können, ohne die Implementierung der Datenverarbeitung verändern zu müssen.

Algorithmen als GraphStream Source

Wir haben uns entschlossen, für eine bessere Visualisierung des Graphen die Streaming Schnittstelle von GraphStream zu verwenden, über die wir Änderungen des Layouts über eine asynchrone Pipe an den JavaFx Viewer übertragen.

Umsetzung der Aspekte der Implementierung



Unsere Applikation ist grob so aufgebaut, wie in der Abbildung. Es gibt eine Hauptklasse, die das Hauptfenster verwaltet. Sie hält auch die Instanz des GraphStream Graphen, auf dem gearbeitet wird. Bei dem Start lädt sie alle Fxml Files, die sie finden kann und initialisiert diese als JavaFx Nodes mit Controller. Über diese Controller werden Eingaben entgegen genommen und Aktionen verwaltet.

Algorithmen

Bei der Implementierung der Algorithmen haben wir uns entschlossen ein abstrake Klasse AbstractAlgorithm einzuführen um Code der sonst redundant geschrieben werden müsste an einer

zentralen Stelle zu behalten. Letztlich werden auf Instanzen von AbstractAlgorithm 3 Methoden aufgerufen: doInitialize(), doNextStep() und doTermination(). Diese Methoden werden in dieser Reihenfolge aufgerufen. doNextStep() wird so lange aufgerufen, bis als Ergebnis false zurückgegeben wird, was bedeutet, dass der Algorithmus fertig ist. Zuletzt wird doTermination() aufgerufen. Wenn eine Klasse von AbstractAlgorithm erbt, müssen die Methoden init(), nextStep() und terminate() implementiert werden.

Die BreadthFirstSearch Algorithmus Klasse basiert auf AbstractAlgorithm. Der Algorithmus findet einen Weg zwischen 2 Punkten und stellt diesen mithilfe der AbstractAlgorithm Klasse visuell dar. Der Algorithmus ist in zwei Teile geteilt. Das Markieren der Knoten, solang bis ein Weg gefunden wurde und dann Rückverfolgung dieses Weges.

Es ist in der Aufgabe vorgegebenen ein Graph aus einer Datei des folgenden Formates zu laden und auch zu schreiben.

```
["#directed;"]
node1,[":"attr1],[","node2,[" :"attr2],[" ("edge")"],[" :: "weight]];
```

`node1`, `node2` und `edge` sind Zeichenketten
`attr1`, `attr2` und `weight` sind Zahlen

Graph Laden

Das Laden des Graphen wird von der Klasse GraphLoader erledigt. Diesem wird bei der Initialisierung eine Instanz der Schnittstelle GraphReceiver übergeben. Dieser Schnittstelle kann übermittelt werden, dass ein Knoten oder eine Kante hinzugefügt wurde. Außerdem kann die Information empfangen werden, dass die folgenden Kanten gerichtet sind oder nicht. Die Klasse GraphStreamGraphReceiver stellt die Verbindung zur GraphStream API her. Sie implementiert sowohl die GraphReceiver Schnittstelle als auch die Source Schnittstelle von Graphstream. Somit werden die Daten der Graph Dateien vom GraphLoader gelesen, der sie an die GraphStreamGraphReceiver Klasse schickt, welche sie dann an den Graphen von GraphStream weiterleitet.

Graph Speichern

Die Schnittstelle GraphWriter bietet mit der Methode writeUnit eine Methode an, eine Zeile des gegeben Datenformats zu schreiben. Es muss mindestens ein Knoten gegeben sein. Alle anderen Werte sind Optional. Diese Schnittstelle wird von der Klasse SimpleGraphWriter implementiert, welchem bei der Initialisierung eine Writer Instanz übergeben wird, in welche die Zeilen des Dateiformates geschrieben werden. Die Klasse GraphStreamGraphPersistor ist dafür da, den GraphStream Graph auf die GraphWriter Schnittstelle abzubilden. Sie nimmt eine Instanz von GraphWriter und einen GraphStream Graph an. Mit der Methode persist werden alle Elemente des Graphen an den GraphWriter geschickt und in den Writer geschrieben.

Testfälle

Laden und Speichern

Für das Laden und speichern haben wir folgenden Testfälle implementiert

Laden:

- Leere Datei → Leerer Graph

- Gerichteter Graph → Alle Kanten gerichtet, Ungerichteter Graph → Keine Kante gerichtet

- Korrekte Datei → Prüfung ob Graph korrekt geladen wurde

- Fehlerhafte Datei → Es wird eine Exception für fehlerhafte Zeile geworfen.

- Gewicht → Prüfen ob korrekt geladen wird

- Attribute von Knoten → Prüfen ob korrekt geladen wird.

Speichern:

- Graphen, wo alle Kanten gerichtet sind → Erste Zeile muss „#directed;“ sein

- Graphen, wo nicht alle Kanten gerichtet sind → Erste Zeile darf nicht „#directed;“ sein

- Leerer Graph → Leere Datei

- Gewicht von Kanten → Prüfen ob korrekt gespeichert wird

- Attribute von Knoten → Prüfen ob korrekt gespeichert wird.

- Knoten mit Knotengrad 0 → Prüfen ob korrekt gespeichert wird

BFS Algorithmus

Für unsere Implementierung des Breadh-First-Algorithmus haben wir folgende Testfälle implementiert.

- Eingegebener Graph darf nicht null sein → Prüfen ob Exception geworfen wird

- Die Werte der Knoten Ids müssen im Graph vorhanden sein. → Wenn nicht null Pointer Exception

- Prüfen ob Markierungen Korrekt sind → Nach beendigung des Graphen Markierungen ausgeben lassen und abgleichen

- Nicht existierende Wege müssen als diese erkannt werden

- Existierende wege müssen erkannt werden.

- Gefundene Wege müssen kürzeste Wege sein