Inhaltsverzeichnis

[GKA-Praktikum1 1](#_Toc465638469)

[Algorithmen 2](#_Toc465638470)

[FileParser 2](#_Toc465638471)

[BFS 2](#_Toc465638472)

[Dijkstra 3](#_Toc465638473)

[DistanzUpdate 4](#_Toc465638474)

[KuerzesterWeg 4](#_Toc465638475)

[Methodik zum Messen der Zugriffe auf den Graphen 5](#_Toc465638476)

[Fragen aus der Aufgabe 5](#_Toc465638477)

[Testkonzept 6](#_Toc465638478)

# GKA-Praktikum1

Es soll eine Anwendung entwickelt werden, mit der man Graphen be- und verarbeiten kann.

Die Anwendung soll:

* Graphen aus Dateien einlesen und auch wieder Speichern können
* Den BFS-Algorithmus implementiert haben
* Den kürzesten weg und die Anzahl der benötigten Kanten liefern
* Die Möglichkeit bieten, Zugriffe auf den Graphen zu Messen

Um dies zu realisieren wurde die Graphstream-Bibliothek gewählt. Da sie die Visualisierung und das Aktualisieren der visuellen Darstellung des Graphen übernimmt, ist es möglich sich auf die Algorithmik der Suche, des Einlesens bzw. speichern der .gka Dateien sowie die Messung der Zugriffe zu Konzentrieren. Die Entwicklung der Anwendung wird von Patrick Höling und Hendrik Seemann geplant und durchgeführt.

Das Projekt wurde wie folgt aufgeteilt:

Einarbeitung in Graphstream: Höling/Seemann

Implementierung der Algorithmen: Höling

JUnit: Seemann

Dokumentation: Höling/Seemann

# Algorithmen

## FileParser

Der FileParser liest die Datei aus und liefert eine Menge an Knoten und eine Menge an Kanten zurück. Die Datei wird zeilenweise ausgelesen. Aus der momentanen Zeile werden mit regulären Ausdrücken die Namen der Knoten, die Kantenausrichtung sowie eventuell vorhandene Kantenbezeichnungen oder Gewichte entnommen.

Parsefile(t)  
Input: Text t ist die eingelesene .gka Datei  
Output: Zwei Sets, einmal alle Knoten, einmal alle Kanten  
  
Mn := (leere) Menge für Knoten  
Me := (leere) Menge für Kanten  
directed := true  
weighted:=false  
WHILE !EOF  
 line := Zeile aus t  
 if line contains “->” then  
 directed := false  
 end if  
 if line contains “:” then  
 weighted:=true

end if  
 Knoten1 := getNode(line)  
 Knoten2 := getNode(line)  
 Richtung := getDirection(line)  
 KantenName := getName(line)  
 Gewicht := getWeight(line)

Mn.add(Knoten1)  
 Mn.add(Knoten2)

Kante := neue Kante(Knoten1, Knoten2, Richtung, KantenName, Gewicht)  
 Me.add(Kante)  
ENDWHILE  
return

## BFS

Die breitensuche durchsucht einen ungewichteten Graphen und stellt fest ob ein punkt erreichbar ist und leistet die Vorarbeit um den kürzesten weg zu finden.

btsSuche(start, ende)  
Input: start ist der Knoten von dem die Suche ausgeht  
 ende ist der Knoten der gesucht wird  
Output: boolean ob ein Punkt erreichbar ist  
  
erfolgreich := dijkstra(start, ende) mit der Einstellung nicht auf die Kantengewichte zu achten  
  
return erfolgreich

## Dijkstra

Dijkstra macht eine breitensuche vom Startknoten zum Zielknoten. Dabei werden die Nachbarpunkte die ich erreichen kann, einer Queue hinzugefügt. Der startknoten wird der Queue hinzugefügt. Dann werden so lang alle Knoten untersucht bis die Queue leer ist. Wenn ein neuer Knoten untersucht wird, werden alle Entfernungen aktualisiert. Das bedeutet alle Knoten die dem momentanen Knoten gegenüberliegen und auch erreichbar sind (gerichtete Kante), werden die neue minimale Entfernung mitgeteilt. Wenn irgendwann der Startknoten untersucht wird, kann der Minimale Weg zurückverfolgt werden.

Dijkstra(start, ende, option)  
Input: start ist der Knoten von dem die Suche ausgeht  
 ende ist der Knoten der gesucht wird  
 Option ist die Option das Kantengewicht zu missachten und mit einem Kantengewicht von   
 1.0 zu rechnen  
Output: liefert ob der Algorithmus den endknoten vom start ausgehend erreichen kann  
  
erfolgreich := false #sagt aus ob der Algorithmus erfolgreich war  
Qn := (leere) Queue in die Knoten für die Breitensuche enthalten sein sollen.  
Qn.add(start)  
  
WHILE |Qn| > 0 ODER   
 Knoten := minimum(Qn) #liefert den Knoten mit der geringsten Entfernung  
 AnzahlKanten := countEdge(Knoten) #Die Anzahl an Kanten die ein Knoten hat  
  
 FOR jede Kante die an dem Knoten anliegt  
 distanzUpdate(Kante, option) # hier liegt der Unterschied zwischen BFS und Dijkstra  
 NachbarKnoten := opposite(Knoten) # der gegenüberliegende Knoten auf der   
 #momentanen Kante  
 IF NachbarKnoten!=besucht UND nicht in Qn  
 Qn.add(NachbarKnoten)  
 END IF

END FOR  
 IF Knoten = ende   
 erfolgreich := true  
 END IF  
   
END WHILE   
IF erfolgreich  
 kuerzesterWeg(start, ende) # hier wird der kürzeste Weg eruiert  
END IF  
return erfolgreich

## DistanzUpdate

Anhand der übergebenen Kante und dem Kantengewicht, werden die Distanzen zwischen den Knoten ermittelt. Wenn die Entfernung des Knotens (der übergeben wurde) addiert mit dem Kantengewicht geringer ist als die Entfernung die der gegenüberliegende Knoten hat momentan hat, wird diese mit dem neuen Minimum überschrieben.

distanzUpdate(**Knoten**, **Kante**, **option**)  
Input: **Knoten** der Knoten dessen kanten untersucht werden  
 **Kante** die Kante die vom Knoten ausgehend ist  
 **option** wenn das tatsächliche Kantengewicht ignoriert werden soll und stattdessen mit   
 einem Gewicht von 1.0 gerechnet werden soll (True = ignorieren, false = tatsächliches   
 Gewicht)

Nachbarknoten := opposite(Kante)  
NachbarGewicht := NachbarKnoten.gewicht()  
  
   
IF option  
 KantenGewicht := 1.0  
ELSE  
 Kantengewicht := Kante.gewicht()  
END IF

neueKosten := knoten.gewicht() + KantenGewicht  
  
IF Knoten != kante.ziel # impliziert das die Kante auch gerihtet sein kann  
 Nachbarknoten.gewicht := minimum(NachbarGewicht, neueKosten)  
END IF

## KuerzesterWeg

Nachdem die BFS Suche oder Dijkstra durchgelaufen sind um den Kanten die passenden Gewichte zu geben, wird am Endpunkt angefangen der kürzeste weg zu suchen. Dazu werden Alle an dem Knoten anliegenden Kanten angesehen und bei dem Knoten der das geringste Gewicht hat, wird erneut gesucht. So lang bis der untersuchte Knoten ein Gewicht von 0.0 hat;

kuerzesterWeg(ende)  
Input: ende ist der Endknoten bei dem die Suche nach dem kürzesten Wegs beginnt.  
Output: eine Liste die den kürzesten weg beginnend am Ende enthält

Le := (leere) Liste in der die Kanten des kürzesten Wegs eingefügt werden  
tempKnoten := ende  
  
  
WHILE tempKnoten.gewicht != 0.0  
 FOR alle Kanten von tempKnoten  
 #nimmt den Knoten mit dem kleinsten gewicht  
 minimumKnoten := minimum(minimumKnoten, opposite(tempKnoten))   
   
 END FOR  
 Le.add(Kante(tempKnoten,minimumKnoten)) #fügt die Kante mit den geringsten Kosten zur   
 #Liste hinzu  
 tempKnoten:=minimumKnoten  
   
  
END WHILE  
return Le

# Methodik zum Messen der Zugriffe auf den Graphen

Um die Zugriffe auf den Graphen zu messen, werden in den Algorithmen einem MeasureObjekt mitgeteilt, welcher Teil des Algorithmus im Source Code grad ausgeführt wird und wie viele schreibende und lesende Zugriffe dieser Teil hat. Dabei wird mit folgender Tabellen gearbeitet:

Lesende Zugriffe (Methodenaufrufe) auf dem Graphen oder eines seiner Bestandteile:

|  |  |
| --- | --- |
| Iterator.next | Wert+1 |
| get\* (sämtliche getter) | Wert+1 |
|  |  |

Schreibende Zugriffe (Methodenaufrufe) auf dem Graphen oder eines seiner Bestandteile:

|  |  |
| --- | --- |
| set\* (sämtliche setter) | Wert+1 |
| remove | Wert+1 |
|  |  |

Dieses MeasureObjekt kann feststellen wie oft ein Algorithmus Zugriffe hatte und in welchen Teil des Source Codes diese waren.

# Fragen aus der Aufgabe

**Was passiert, wenn Knotennamen mehrfach auftreten?**Da beim Auslesen der Datei alle entdeckten Knoten in eine Menge gelegt werden, kommen Dubletten nicht vor. Sollte es also beabsichtigt sein Knoten mehrfach vorkommen zu lassen, werden die Dubletten als 1 knoten behandelt.

**Wie unterscheidet sich BFS für gerichtete und ungerichteten Graphen?**Bei gerichteten Graphen muss zusätzlich drauf geachtet werden, das der kürzeste Weg auch erreichbar ist. Diese Logik muss mit in den Algorithmus eingearbeitet werden.

**Wie können Sie testen, dass ihre Implementierung auch für sehr große Graphen funktioniert?**Eine Möglichkeit wäre einen Generator zu entwickeln, der eine der eine „sehr große“ Anzahl an Knoten und Kanten generiert. Anschließend werden die Algorithmen getestet, Eine andere Möglichkeit wäre, anzunehmen das die Implementierung dem gewachsen ist, da sie schon bei kleineren Graphen funktioniert.

# Testkonzept