Inhaltsverzeichnis

[GKA-Praktikum1 1](#_Toc467436429)

[Algorithmen 1](#_Toc467436430)

[Dijkstra 1](#_Toc467436431)

[DistanzUpdate 2](#_Toc467436432)

[KuerzesterWeg 3](#_Toc467436433)

[Methodik zum Messen der Zugriffe auf den Graphen 3](#_Toc467436434)

[Fragen aus der Aufgabe 4](#_Toc467436435)

[Testkonzept 4](#_Toc467436436)

# GKA-Praktikum2

Aufbauend auf der zuvor entwickelten Anwendung sollen nun weitere Algorithmen entwickelt werden. Die Algorithmen sind Djikstra und Fyord-Warshall. Es sollen bei der Ausführung jeweils die Zugriffe auf den Graphen gemessen werden.

Das Projekt wurde wie folgt aufgeteilt:

Entwicklung Generator: Höling/Seemann

Implementierung der Algorithmen: Höling/Seemann

JUnit: Höling/Seemann

Dokumentation: Höling/Seemann

# Algorithmen

## 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

# Flyod-Warshall

Der Algorithmus Analysiert schrittweise die Pfade zwischen allen Paaren von Knoten und nur die besten Wege werden gespeichert.

FloydWarshall(Graph, Start, Ende)  
Input: Graph ist der Graph auf dem die Operationen stattfinden  
 Start ist der start Knoten der Suche  
 Ende ist das Ziel der Suche nach dem kürzesten Weg.

d := ist eine Matrix die die Distanzen der Knotenpaare beinhaltet. Ist |Anzahl der Knoten|x|Anzahl   
 der Knoten|groß  
  
Initialisierung()  
FOR j von 1 bis |Anzahl der Knoten|  
 FOR i von 1 bis |Anzahl der Knoten| wobei i!=j  
 FOR k von 1 bis |Anzahl der Knoten|wobei k!=j  
 Momentangewicht := dik  
 neues Gewicht := minimum(dik, (dij+djk))  
 IF momentangewicht != neuesgewicht  
 tik := j  
 dik := neues Gewicht  
 END IF  
 END FOR  
 END FOR  
END FOR

## Initialisierung

Initialisiert die Matrizen für den FloydWarshall  
  
initialize()  
Input : keiner  
Output: keiner

d := ist die Distanzmatrix in der die Distanzen der Knotenpaare gehalten werden  
t := ist die Transitmatrix in der die Vorgänger gehalten werden sollen

FOR i bis |Anzahl der Knoten|  
 FOR j bis |Anzahl der Knoten|  
 IF i==j  
 dij := 0.0  
 tij := i  
 ELSE   
 dij := UNENDLICH  
 tij := -1  
 END IF  
 END FOR  
END FOR

WHILE für jede Kante im Graphen  
 s := Quellknoten einer Kante  
 t := Zielknoten einer Kante  
 is := Index des Quellknotens auf der Matrix  
 it := Index des Zielknotens auf der Matrix

dis,it := Gewicht der Kante  
 tis,it := is  
END WHILE

# 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

**Bekommen Sie für einen Graphen immer den gleichen kürzesten Weg? Warum?**

**Was passiert, wenn der Eingabegraph negative Kantengewichte hat?**Bei dem Algorithmus FyordWarshall wird festgestellt, wenn bei negativen Kantenkosten die Summe eines Paares kleiner als 0 ist, dass ein Kreis vorhanden sein muss.

**Wie allgemein ist Ihre Konstruktion von BIG, kann jeder beliebige, gerichtete Graph erzeugt werden?**Ja, es werden zufällig Kanten zwischen zufällig ausgewählten Knoten erzeugt.

**Wie testen Sie für BIG, ob Ihre Implementierung den kürzesten Weg gefunden hat?**Es wird in BIG ein kürzester weg vordefiniert. Die Algorithmen sollten genau diesen finden.

**Wie müssten Sie Ihre Lösung erweitern, um die Menge der kürzesten Wege zu bekommen?**Die Knoten müssten in eine HashMap<Key, Value> gespeichert werden. Die Keys sind die Knoten die eine Kante eines kürzesten Weges enthalten und die Values sind die adjazenten Knoten die auf einem kürzesten Weg liegen.

**Wie müssten Sie Ihre Lösung erweitern, damit die Suche nicht-deterministisch ist?**

# Testkonzept