BAI3-GKA WiSe2§
Graphentheoretische Konzepte und Algorithmen

Praktikumsaufgabe-Template

Deckblatt

Geben Sie bitte Ihre Namen, Ihr Team und die Gruppe an:

GKA-Gruppe	
Team	
	Ihren Namen hier ersetzen
	Ihren Namen hier ersetzen
	Ihren Namen hier ersetzen

Bearbeitete Themen in Stichpunkten:

Name	
Name	
Name	

Geschätzte Arbeitszeiten in Stunden:

Name	
Name	
Name	

1 Einleitung

1.1 Dijkstra Algorithmus in Worten

Wie genau funktioniert der Dijkstra Algorithmus und wo wird er angewendet? Nehmen wir an Kreuzungen und Kreisverkehre seien Knoten und die Straßen, die diese verbinden die Kanten. Dann lässt sich bis auf Ausnahmen jedes Straßennetz durch einen Graphen darstellen. Um zu berechnen, wie Mensch am schnellsten von einem beliebigen Kreisverkehr zu einer beliebigen Kreuzung kommen kann, wird nun zum Beispiel der Dijkstra Algorithmus verwendet.

1. Finden des Startknotens im Graphen.
2. Dann die Distanz zu allen adjazenten Knoten berechnen und diese zusammen mit der Distanz zum Startknoten und den Knoten auf dem Weg dorthin in einer Open List speichern.
3. Den Ausgangsknoten auf eine Closed List verschieben, da alle inzidenten Kanten bearbeitet wurden.
4. Nun den Knoten mit kürzester Distanz zum Startknoten in der Open List suchen und Schritt 2 von diesem Knoten aus wiederholen
   (a) Wenn ein Knoten, welcher bereits in der Open List gespeichert ist, erneut besucht wird, dann wird verglichen ob die neue Distanz kürzer ist. Wenn ja, wird der Knoten inklusive der Distanz zum Startknoten und den Knoten auf dem Weg dorthin aktualisiert. Wenn die Distanz länger ist, wird der Weg verworfen.
   (b) Wenn alle inzidenten Kanten eines Knotens bearbeitet wurden, wird dieser Knoten in die Closed List verschoben.
5. Der Algorithmus ist beendet, wenn entweder
   (a) alle Knoten des Graphen besucht wurden oder
   (b) die Distanz vom Start- zum Zielknoten, die bisher kleinste gefundene Distanz ist.

1.2 Wählen des Grundansatzes

Zur objektorientierten Abstrahierung eines Graphen gibt es zwei verschiedene Ansätze. Der erste Ansatz wäre, dass es einen Typ *Kante* gibt, in welchem die vor- und nachfolgenden Knoten und die Länge der jeweiligen Kante gespeichert werden.
Diese Kanten würde dann den Graphen bilden und diesen in Form eines Sets darstellen:
$Set < Kante > graph = new\ HashSet <> ()$.

Der zweite Ansatz wäre die Kante als solches zu abstrahieren und nicht als eigenen Typen zu implementieren. Es gäbe dann nur den Typen Knoten, welcher alle benachbarten Knoten mit zugehöriger Distanz enthält.
Auch diese Knoten können nun in einem Set gespeichert werden:
$Set < Knoten > graph = new\ HashSet <> ()$.

Die zu verwendende Java Bibliothek Graphstream enthält einen Typen Multigraph, welcher den ersten Ansatz implementiert.
Eine Beispiel Implementierung des zweiten Ansatzes ist im folgenden Blog Eintrag[1] implementiert.

[1] `https://www.baeldung.com/java-dijkstra`

1.3 Aufbau des Graphen

Um den Dijkstra Algorithmus innerhalb eines Graphen anwenden zu können, muss zuerst ein Graph aufgebaut werden. Wie im vorherigen Kapitel beschrieben, genügt es einen Multigraphen zu definieren und diesem Kanten hinzuzufügen. Sowohl die Kanten als auch die Knoten müssen einen eindeutigen, im Graphen nur einmal vorkommenden Namen haben.

Bedingungen, welche für den Graphen gelten:

- Der Graph muss entweder komplett gewichtet sein und dann ausschließlich positive Kantengewichte haben und komplett ungewichtet sein.
- Der Graph kann unendlich viele Knoten besitzen.
- Der Graph kann gerichtet, ungerichtet oder gemischt sein.

1.4 Open und Closed List

Knoten in der Closed List dienen der Kontrolle ob alle Kanten eines neu gefundenen Knoten bereits besucht wurden. Wird ein neuer Knoten gefunden, welcher noch nicht in der Open List enthalten ist, muss nun kontrolliert werden ob dieser bereits vollständig abgearbeitet ist, d.h. ob alle inzidenten Kanten bereits geprüft wurden.
Es reicht ein einfaches Set, welches die bereits besuchten Knoten enthält: $Set < Knoten > closedList$. Durch die Verwendung eines Sets wird außerdem das mehrfache Hinzufügen eines Knotens verhindert.

In der Open List wird der derzeitige Fortschritt festgehalten. Die verschiedenen Elemente können als Tabelleneintrag gespeichert werden. Eine Zeile dieser als Tabelle dargestellten Open List könnte folgendermaßen definiert werden:

Beschreibung	Knoten	Länge zum Startknoten	Knoten auf dem Weg
Typ	Node	Integer	$List < Knoten >$
Beispiel	"D"	44	{{"A", ... },{"C", ... }}

Tabelle 1: Open List

1.4.1 Open List als Set

Auch wenn es dem Namen widerspricht, kann die Open List als Set implementiert werden. Wenn ein Knoten hinzugefügt wird, muss erst geschaut werden ob dieser schon vorhanden ist. Zum Vereinfachen dieser Suche wird eine Map genutzt: $Map < Knoten,\ Weg > openList$, wobei der Weg die Distanz zum Startknoten und die Knoten auf diesem Weg enthält. Wenn der Knoten nicht vorhanden ist, wird der Map ein neues Element hinzugefügt, welches als Key den neuen Knoten und als Value dessen Distanz zum Startknoten und den Knoten auf diesem Weg enthält.
Nach jeder Iteration muss der Knoten aus der Open List mit dem kürzestem Weg zum Startknoten bestimmt werden. Mit dieser Struktur ist dieser Prozess mit viel Rechenaufwand verbunden.

1.4.2 Open List als Priority Queue

Java bietet einen Typen Priority Queue an. Diesem kann eine Comparator übergeben werden, welcher die in der Queue enthaltenden Elemente effizient nach bestimmten Attributen sortiert.
So können zum Beispiel verschiedene Listen mit Knoten beim Einfügen von Elementen direkt nach der in den Knoten enthaltenden Länge zum Startpunkt sortiert werden.
Ein großer Vorteil, den diese Implementierung mit sich bringt ist, dass durch die Funktion

PriorityQueue.peak() effizient der Knoten für die Wiederholung des Algorithmus bestimmt werden kann.
Der Typ Knoten muss des weiteren nur noch die adjazenten Knoten mit der Distanz enthalten.

2 Dokumentation der Implementierung

2.1 Einführung neuer Typen

Wie bereits in 1.4.2 beschrieben, ist die Implementierung der Open List als PriorityQueue am sinnvollsten.

Zur effizienteren Sortierung wird einer neuer Typ *PriorityQueueItem* eingeführt.

```
@Data
@NoArgsContructor
public class PriorityQueueItem {

    /* Um das Item bei falscher Initialisierung nicht auf die
    erste Position in der PriorityQueue zu schieben, wird es mit einer
    maximalen Distanz initialisiert.*/
    private int distance = Integer.MAX_VALUE;

    /*Die Reihenfolge dieser Knoten muss beibehalten werden,
    da ansonsten der kuerzeste Weg nachtraeglich nicht zurueckgegeben
    werden kann*/
    private List<Node> nodes = new LinkedList<>(); //Node -> Knoten
}
```

2.2 Ablauf des Algorithmus mit einer Priority Queue

1. Ein leeres $Set < Node >$ *closedList* und eine leere $PriorityQueue < PriorityQueueItem >$ *priorityQueue* werden initialisiert.
2. Für den Startknoten wird ein *PriorityQueueItem* initialisiert und der *priorityQueue* hinzugefügt. Dieses hat eine *distance* von 0 und der *nodes* Liste wird der Startknoten hinzugefügt.
3. Jeder adjazente Knoten des Ausgangsknotens wird betrachtet:
   (a) Ist der Knoten bereits in der closedList: Knoten muss nicht weiter beachtet werden.
   (b) Für jeden weiteren wird ein neues *PriorityQueueItem* initialisiert. Die *distance* ist die Distanz zwischen dem Ausgangsknoten und dessen adjazentem Knoten. Die *nodes* Liste enthält den neuen Knoten.
   Diese neuen *PriorityQueueItems* werden nun jeweils mit dem bereits bestehendem *PriorityQueueItem* des Ausgangsknotens ergänzt. Hierbei werden die Distanzen aufsummiert und die Listen konkateniert (Reihenfolge beachten).
4. Die neu entstandenen *PriorityQueueItems* ersetzen nun das *PriorityQueueItem* des Ausgangsknotens in der *PriorityQueue* unter folgender Bedingung:
   (a) Ist der letzte Knoten aus der *nodes* Liste einer dieser *PriorityQueueItems* bereits das letzte Element einer *nodes* Liste eines *PriorityQueueItems* in der *PriorityQueue* muss verglichen werden, welche *distance* der *PriorityQueueItems* kleiner ist und nur dieses *PriorityQueueItem* wird behalten.

5. Da alle Kanten des Ausgangsknotens jetzt untersucht wurden, wird dieser der *closedList* hinzugefügt.

6. Das oberste Element der *PriorityQueue* wird bestimmt. Das letzte Element aus dessen *nodes* Liste wird zum neuen Ausgangsknoten und Schritt 3 wird wiederholt.

7. Der Algorithmus ist beendet, wenn entweder

   (a) alle Knoten des Graphen besucht wurden oder

   (b) der Zielknoten das letzte Element der *nodes* Liste eines bereits in der *PriorityQueue* enthaltendem *PriorityQueueItem* und die *distance* die bisher kleinste gefundene Distanz ist.

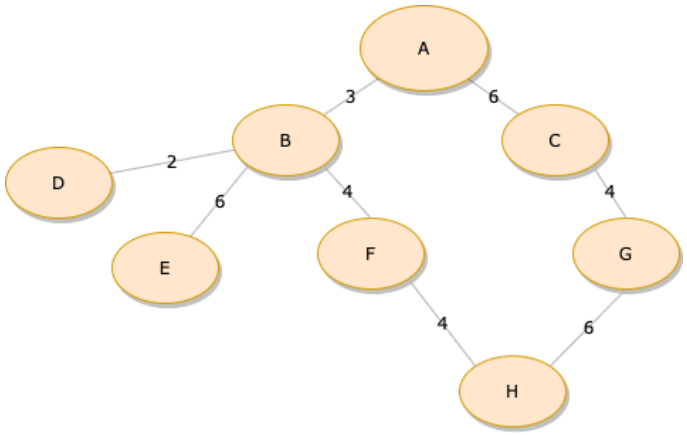

Abbildung 1: Gewichteter, ungerichteter Graph

2.2.1 Beispiel

Sei der Knoten "A" auf folgendem Graphen 1 ein Startknoten. Nach zwei Iterationen sieht der Inhalt der PriorityQueue wie folgt aus:

```
priorityQueue = {
    {5, {A, B, D}},
    {6, {A, C}},
    {7, {A, B, F}},
    {9, {A, B, E}}
};

closedList = { };
```

3 Nachwort

3.1 Dijkstra als Baum

Eine weitere Möglichkeit den Algorithmus zu implementieren wäre es die besuchten Knoten als Baum nachzubauen, während der originale Graph bearbeitet wird. Der Startknoten ist nun der Elternknoten, während nach der ersten Iteration alle adjazenten Knoten die Kinderknoten sind.

[KN12] und Ihre Abbildungen nicht, z.B. Abb. **??**.

4 Beantwortung der Fragen

1. Antwort auf die erste Frage
2.

Literatur

[KN12] Sven Oliver Krumke and Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen, 3. Auflage*. Leitfäden der Informatik. Teubner, 2012.