**LK-Q1-V**

**2. Die Datenstruktur Graph im Anwendungskontext unter Nutzung der Klassen Graph, Vertex und Edge.**

(d) Bestimmung kürzester Wege in Graphen im Anwendungskontext (Backtracking, Dijkstra-Algorithmus)



**Aufgabe**

*Entwickeln Sie einen Algorithmus, der den kürzesten Weg von einem Startknoten S zu einem Zielknoten Z in einem gegebenen Graphen bestimmt.*

1. Backtracking

Anknüpfend an den Backtracking-Algorithmus zur Suche aller Weg in einem Graphen, findet folgender Algorithmus sicher den kürzesten Weg:

Bestimme alle Wege von S nach Z

Bestimme die Länge dieser Wege

Suche den Weg mit der kleinsten Länge

Etwas schneller funktioniert es, wenn der Algorithmus zur Bestimmung aller Wege die Weglaenge mitberechnet und sich den bis dahin kürzesten Weg merkt. In der folgenden Java-Methode sind die Unterschiede im Quelltext zwischen den Methoden sucheAlleWege und sucheKuerzestenWeg grau schraffiert.

public double sucheKuerzestenWeg(Vertex knoten, Vertex ziel, List<Vertex> weg,

List<Vertex> kurzWeg, double wegLaenge, double minWegLaenge){

knoten.setMark(true);

weg.append(knoten);

if (knoten != ziel){

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (nachbarn.hasAccess()) {

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()){

double distanz = bspGraph.getEdge(knoten, nachbar).getWeight();

wegLaenge = wegLaenge + distanz;

minWegLaenge = sucheKuerzestenWeg(nachbar, ziel, weg, kurzWeg,

wegLaenge, minWegLaenge);

//Schritt rückgängig machen

weg.toLast();

Vertex loeschKnoten = weg.getContent();

loeschKnoten.setMark(false);

wegLaenge = wegLaenge - distanz;

weg.remove();

}

nachbarn.next();

}

} else {

if (wegLaenge < minWegLaenge){

minWegLaenge = wegLaenge;

copy(weg, kurzWeg);

}

}

return minWegLaenge;

}

**Aufgabe**

*Analysieren Sie die Methode sucheKuerzestenWeg mit dem Startknoten I und dem Zielknoten L. Geben Sie die Inhalte der Liste kurzWeg sowie die zugehörigen Werte von minWegLaenge an, die der Algorithmus generiert.*

Lösung:

|  |  |
| --- | --- |
| kurzWeg | minWegLaenge |
| I G A B C E L | 260 |
| I G F B C E L | 245 |
| I G F E L | 195 |
| I G K E L | 175 |
| I K E L | 110 |

**Aufgabe**

*a) Bestimmen Sie die Anzahl der Wege von einem beliebigen Start- zu einem beliebigen Zielknoten in Graphen mit 2, 3, 4, 5 Knoten, wenn alle Knoten durch Kanten paarweise miteinander verbunden sind.*

*b) Leiten Sie eine Formel für n Knoten her.*

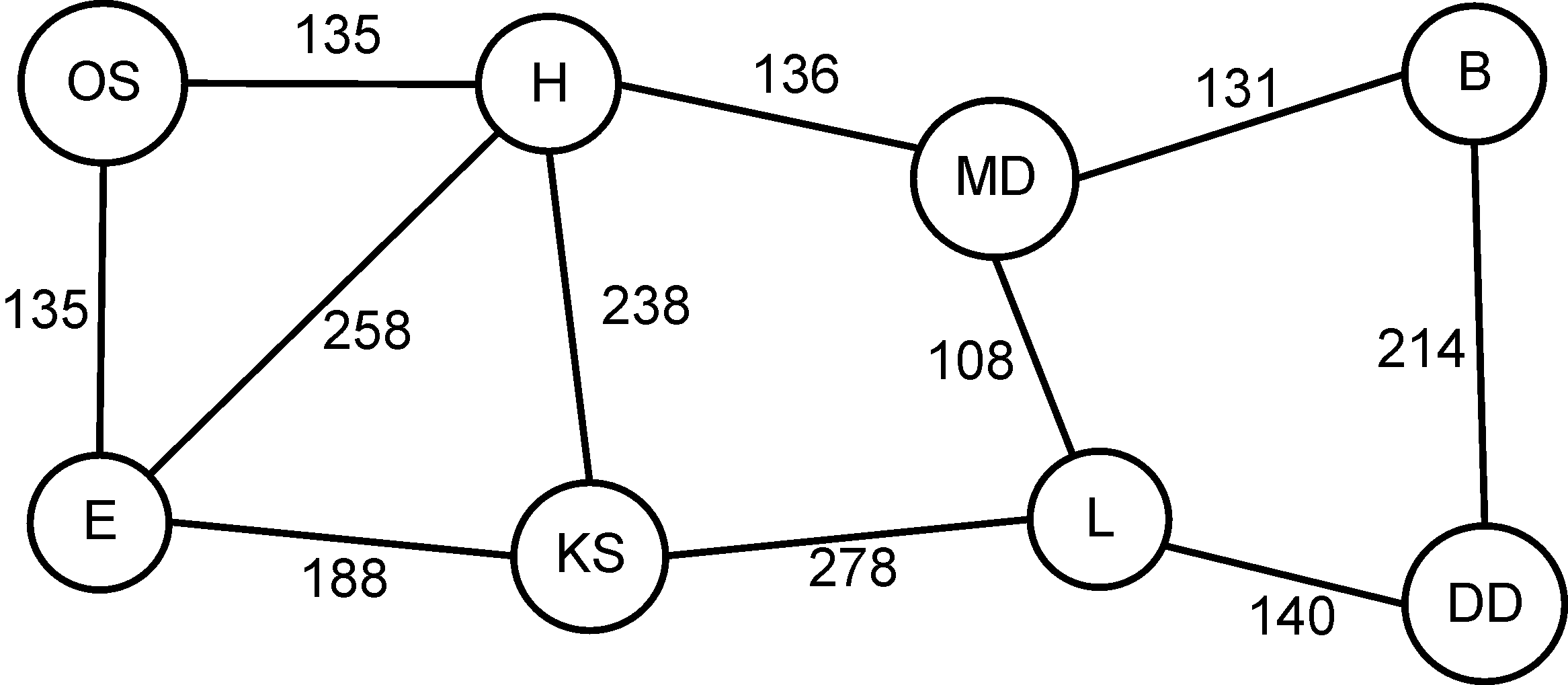
*c) Begründen Sie, warum das Backtracking-Verfahren zur Bestimmung des kürzesten Weges zwischen zwei Knoten in einem Graphen für Navigationssysteme nicht geeignet ist.*

**2. Dijkstra-Algorithmus**

Einen weitaus schnelleren Algorithmus hat Edsger W. Dijkstra entwickelt. Mit folgender Aufgabe wird der Algorithmus erarbeitet.

**Aufgabe**

Ein reduziertes Straßen-/Autobahnnetz zwischen acht deutschen Großstädten lässt sich verein- facht durch folgenden Graphen darstellen.



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| B | DD | E | H | KS | L | MD | OS |
| Berlin | Dresden | Essen | Hannover | Kassel | Leipzig | Magdeburg | Osnabrück |

B Berlin DD Dresden E Essen H Hannover KS Kassel

L Leipzig MD Magdeburg OS Osnabrück

1. *Entwickeln Sie für obigen Graphen die Adjazenzmatrix. Hinweis: Um die Indizes der einzelnen Städte zu ermitteln, überführen Sie diese einfach in alphabetischer Reihenfolge in Zahlen:*

*B → 1, DD → 2, . . .*

1. Von einem Logistikzentrum in Leipzig sollen Routen zu allen anderen Städten ermittelt werden. Führen Sie deshalb dort beginnend eine vereinfachte Breitensuche durch. Die Vereinfachung besteht darin, dass keine Stadt doppelt in der Schlange der noch nicht besuchten Knoten vorkommen soll.

*(1) Ermitteln Sie die Schlange der noch nicht besuchten Städte.*

*(2) Geben Sie an den Städten im Graphen die Besuchsreihenfolge an und markieren Sie die gewählten Straßen farbig.*

*(3) Stellen Sie in der folgenden Tabelle die ermittelte Entfernung jeder Stadt entlang der markierten Wege nach Leipzig dar.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ort | B | DD | EE | H | KS | L | MD | OS |
| Entfernung nach L |  |  |  |  |  |  |  |  |

1. *Beurteilen Sie, ob die Entfernungen von Leipzig zu allen anderen Städten minimal sind.*
2. Zur Berechnung kürzester Wege ist die Auswahl der zu besuchenden Städte entscheidend.

*a) Erläutern Sie, wie sich bei der Breitensuche die Reihenfolge der zu besuchenden Städte in der Schlange ergibt.*

*b)   Entwickeln Sie ein Verfahren für die Auswahl der nächsten Stadt bei der Breitensuche, um einen möglichst kurzen Weg zu finden.*

*c)   Ermitteln Sie die Wege und deren Längen, die sich bei Anwendung des Verfahrens von b) ergeben.*

1. Um nach Durchführung des Verfahrens die Wege zu allen Städten angeben zu können, speichert man auch bei diesem Verfahren für jede Stadt jeweils den direkten Vorgänger auf dem Weg dorthin.

*Stellen Sie in der folgenden Tabelle die Vorgänger jeder Stadt auf dem kürzesten Weg von Leipzig dar.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ort | B | DD | EE | H | KS | L | MD | OS |
| Vorgänger |  |  |  |  |  |  |  |  |

**Der Dijkstra-Algorithmus**

In Wikipedia wird der Dijkstra-Algorithmus wie folgt beschrieben:

„Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus verspricht. Andere Kanten werden erst dann verfolgt, wenn alle kürzeren Streckenabschnitte beachtet wurden. Dieses Vorgehen gewährleistet, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existieren kann. Eine einmal berechnete Distanz zwischen dem Startknoten und einem besuchten Knoten wird nicht mehr geändert. Distanzen zu noch nicht abgearbeiteten Knoten können sich hingegen im Laufe des Algorithmus durchaus verändern, nämlich verringern. Dieses Vorgehen wird fortgesetzt, bis die Distanz des Zielknotens berechnet wurde (*single-pair shortest path*) oder die Distanzen aller Knoten zum Startknoten bekannt sind (*single-source shortest path*).“[[1]](#footnote-1)

Wie der Beschreibung zu entnehmen ist, benötigt jeder Knoten die Distanz zum Startknoten als weiteres Attribut. Um nach Abarbeitung des Algorithmus’ den Weg wie bei der Breitensuche vom Ziel zum Startknoten generieren zu können, erhält jedes Knotenobjekt als weiteres Attribut einen Verweis auf seinen Vorgänger. Der Graph enthält zur Anwendung des Dijkastra-Algorithmus’ Knoten-Objekte folgender Klasse.

public class DijkstraVertex extends Vertex {

private double distanz;

private DijkstraVertex vorgaenger;

public DijkstraVertex(String pID, int xPos, int yPos){

super(pID, xPos, yPos);

vorgaenger = null;

distanz = -1;

}

//getter und setter

}

Der Algorithmus soll am Beispiel der Wegsuche von I nach L im Beispielgraphen erläutert werden.

Die Prioritätenschlange enthält die noch nicht bearbeiteten Nachbarknoten von bereits bearbeiteten (markierten) Knoten aufsteigend nach ihrer Entfernung zum Startknoten. Die bereits bearbeiteten Konten sind grau markiert.



0. Initialisierung

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, -1, -- | F, -1,-- | G, -1, -- | H, -1, -- | I, -1, -- | K, -1, -- | L,-1,-- |

prioritaetenSchlange

|  |
| --- |
|  |

1. Schritt: Startknoten in Schlange einfuegen

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, -1, -- | F, -1, -- | G, -1, -- | H, -1, -- | I, 0, -- | K, -1, -- | L, -1, -- |

Prioritätenschlange

|  |
| --- |
| I, 0, -- |

2. Schritt: I aus Schlange entfernen, Nachbarknoten nach aufsteigender Entfernung zum Startknoten in Schlange einfügen

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, -1, -- | F, -1, -- | G, 15, I | H, 45, I | I, 0, -- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| K, 10, I | G, 15, I | H, 45, I |

3. Schritt: Ersten Knoten K aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 45, I | I, 0, -- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |  |
| --- | --- | --- | --- |
| G, 15, I | H,45,I | F, 56, K | E, 80, K |

4. Schritt: Ersten Knoten G aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140,G | B, -1, -- | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0, -- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |  |
| --- | --- | --- | --- |
| H, 30, G | F, 56, K | E, 80, K | A, 140, G |

5. Schritt: Ersten Knoten H aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140, G | B, -1, -- | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| F, 56, K | E, 80, K | A, 140, G |

6. Schritt: Ersten Knoten F aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140, G | B, 86, F | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| E, 80, K | B, 86, F | A, 140, G |

7. Schritt: Ersten Knoten E aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140, G | B, 86, F | C, 120, E | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |  |  |  |
| --- | --- | --- | --- |
| B, 86, F | L, 110, E | C, 120, E | A, 140, G |

8. Schritt: Ersten Knoten B aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 106, B | B, 86, F | C, 116, B | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| A, 106, B | L, 110, E | C, 116, B |

8. Schritt: Ersten Knoten A aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 106, B | B, 86, F | C, 116, B | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |  |
| --- | --- |
| L, 110, E | C, 116, B |

9. Schritt: Ersten Knoten L aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 106, B | B, 86, F | C, 116, B | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |
| --- |
| C, 116, B |

Da der Zielknoten L erreicht wurde, kann der Algorithmus abgebrochen werden. Die kürzeste Entfernung vom Start- zum Zielknoten ist als distanz im Knoten L gespeichert, hat also den Wert 110. Der Weg lässt sich ausgehend vom Zielknoten über die Verweise auf die Vorgängerknoten erzeugen.

Als kürzester Weg in umgekehrter Reihenfolge ergibt sich damit die Knotenfolge; L -> E -> K -> I

Bricht man den Algorithmus nicht ab, wenn der Zielknoten erreicht ist, sondern erst, wenn die Prioritätenschlange leer ist, enthalten alle Dijkstra-Knoten des Graphen die kürzesten Entfernungen zum Startknoten, wenn es einen Weg gibt.

**Aufgabe**

*Ermitteln Sie den kürzesten Weg im Beispielgraphen von A nach H mithilfe des Dijkstra-Algorithmus’.*

**Java Methode für den Dijkstra-Algorithmus**

private void dijkstra(DijkstraVertex pStart, DijkstraVertex pZiel){

if (pStart!= null && pZiel != null){

PrioritaetsSchlange schlange = new PrioritaetsSchlange();

pStart.setzeDistanz(0);

schlange.fuegeEin(pStart);

DijkstraVertex nachbarKnoten = null;

DijkstraVertex ersterKnoten = null;

while (!schlange.isEmpty() && ersterKnoten != pZiel) {

schlange.toFirst();

ersterKnoten = (DijkstraVertex)schlange.getObject();

schlange.remove();

if (!ersterKnoten.isMarked()){

ersterKnoten.setMark(true);

List<Vertex> nachbarListe = bspGraph.getNeighbours(ersterKnoten);

nachbarListe.toFirst();

while (nachbarListe.hasAccess()){

nachbarKnoten = (DijkstraVertex) nachbarListe.getContent();

Edge kante = bspGraph.getEdge(nachbarKnoten, ersterKnoten);

double gewicht = kante.getWeight();

//Nachbarknoten in Schlage einfügen oder – falls vorhanden –

//Entfernung ggf. nach unten korrigieren

if ( nachbarKnoten.gibDistanz() == -1 ||

nachbarKnoten.gibDistanz() > ersterKnoten.gibDistanz()+gewicht){

nachbarKnoten.setzeDistanz(ersterKnoten.gibDistanz()+gewicht);

nachbarKnoten.setzeVorgaenger(ersterKnoten);

schlange.fuegeEin(nachbarKnoten);

//korrigierter Knoten wird überschrieben

}

nachbarListe.next();

}

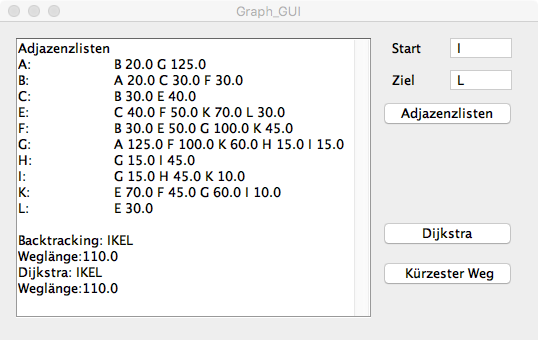
}

}

}

}

Vollständig implementiert sind die beiden Algorithmen zur Suche des kürzesten Weges in einem Graphen im BlueJ-Projekt KuerzesterWeg. Aus Übersichtlichkeitsgründen wurden nicht alle möglichen Fehler abgefangen.



**Aufwand**

Die Laufzeit des Dijkstra-Algorithmus’ ist proportional (n+m)\*log(n), wenn n die Anzahl der Knoten und m die Anzahl der Kanten im Graphen ist, wächst also bei zunehmender Komplexität des Graphen nur sehr langsam an. Daher kann der Dijkstra-Algorithmus zur Berechnung von Weglängen in Navigationssystemen sehr effektiv genutzt werden.

1. de.wikipedia.org Dijkstra-Algorithmus (24.08.2015) [↑](#footnote-ref-1)