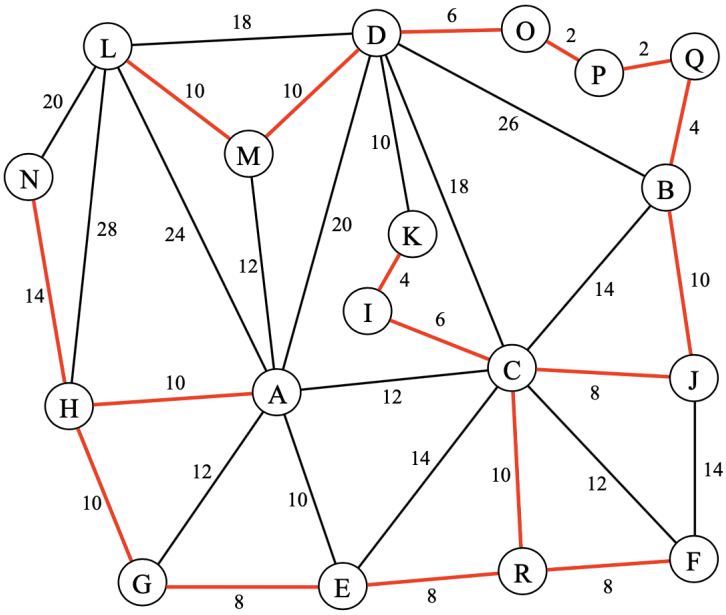
Praktische Informatik II
Tutor*in: Tim Sperling

SoSe 2024
Bearbeiter*in: Simon Berg, Mustafa Erdogan

Übungsblatt 06

Lösungsvorschlag

Aufgabe 1 Spannende Vernetzung



gewählter Teilbaum	gewählte Kante
N	N ⟷ H
N ⟷ H	H ⟷ A
N ⟷ H ⟷ A	H ⟷ G
N ⟷ H ⟷ G	G ⟷ E
N ⟷ H ⟷ G ⟷ E	E ⟷ R
N ⟷ H ⟷ G ⟷ E ⟷ R	R ⟷ F
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷F	R ⟷ C
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C	C ⟷ I
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ I	I ⟷ K
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ I ⟷ K	C ⟷ J
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J	J ⟷ B
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J ⟷ B	B ⟷ Q
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J ⟷ B ⟷ Q	Q ⟷ P
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J ⟷ B ⟷ Q ⟷ P	P ⟷ O
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J ⟷ B ⟷ Q ⟷ P ⟷ O	O ⟷ D
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J ⟷ B ⟷ Q ⟷ P ⟷ O ⟷ D	D ⟷ M
N ⟷ H ⟷ G ⟷ E ⟷ R ⟷ C ⟷ J ⟷ B ⟷ Q ⟷ P ⟷ O ⟷ D ⟷ M	M ⟷ L

Aufgabe 2 Sommer, Sonne, Routenplaner

Aufgabe 2.1 Karte aufbauen

Die Klasse Map soll so erweitert werden, dass ihr Konstruktor die beiden Dateien nodes.txt und edges.txt einliest und daraus einen Graphen konstruiert. Hierzu sollen die bereitgestellten Klasse Node und Edge benutzt werden.

Zunächst wird ein Attribut „nodes “ deklariert, welches die Knoten des Graphen in Form einer ArrayList speichert.

```
22      /**
23       * Liste der Knoten des Graphen
24       */
25      List<Node> nodes = new ArrayList<>();
```

Im Konstruktor werden Daten-Inputstreams erzeugt, um die Textdateien einzulesen, wobei die Zeilen der Textdateien als Strings in ArrayLists abgespeichert werden. Im Falle, dass die Textdateien nicht gefunden oder eingelesen werden können, werden jeweils FileNotFoundExceptions oder IllegalArgumentExceptions geworfen.

```
28      /**
29       * Konstruktor. Liest die Karte ein.
30       *
31       * @throws FileNotFoundException Entweder die Datei "nodes.txt" oder die
32       *                               Datei "edges.txt" wurden nicht gefunden.
33       * @throws IOException           Ein Lesefehler ist aufgetreten.
34       */
35      Map() throws FileNotFoundException, IOException {
36          // Lest hier die beiden Dateien nodes.txt und edges.txt ein
37          // und erzeugt daraus eine Karte aus Node- und Edge-Objekten
38          // Verbindungen sollen immer in beide Richtungen gehen, d.h.
39          // ihr braucht zwei Edges pro Zeile aus der edges.txt.
40          final List<String> fileEdges = new ArrayList<String>();
41          final List<String> fileNodes = new ArrayList<String>();
42
43          try (final BufferedReader stream = new BufferedReader(new InputStreamReader(new
              FileInputStream("edges.txt")))) {
44              String line;
45              while ((line = stream.readLine()) != null) {
46                  fileEdges.add(line);
47              } //gegebenfalls catch weglassen, wieso IllegalArgumentExeption, wenn es kein
                 Argument gibt :o|
48          } catch (final FileNotFoundException e) {
49              throw new IllegalArgumentException("'edges.txt' wurde nicht gefunden.");
50          } catch (final IOException e) {
51              throw new IllegalArgumentException("Ein Lesefehler ist aufgetreten.");
52          }
53
54          try (final BufferedReader stream = new BufferedReader(new InputStreamReader(new
              FileInputStream("nodes.txt")))) {
55              String line;
56              while ((line = stream.readLine()) != null) {
57                  fileNodes.add(line);
58              }
59          } catch (final FileNotFoundException e) {
60              throw new IllegalArgumentException("'nodes.txt' wurde nicht gefunden.");
61          } catch (final IOException e) {
62              throw new IllegalArgumentException("Ein Lesefehler ist aufgetreten.");
63          }
```

Im Anschluss wird der eigentliche Graph erzeugt. Dazu iteriert man als erstes über die Liste der „fileNodes “ und trennt jeden einzelnen String einer Zeile in Substrings auf, die jeweils die Knoten-ID, sowie dessen x- und y-Koordinate repräsentieren. Diese Substrings werden dann in geeignete Datentypen wie int und doubles umgewandelt, um daraufhin neue Knoten mit genau diesen Parametern zu erzeugen und sie der Liste „nodes “ hinzuzufügen.

```
70          for (int i = 0; i < fileNodes.size(); i++) {
71
72              String[] nodeParams = fileNodes.get(i).split(" ");
73
74              int id = Integer.parseInt(nodeParams[0]);
75              double xNode = Double.parseDouble(nodeParams[1]);
76              double yNode = Double.parseDouble(nodeParams[2]);
77
78
79              nodes.add(new Node(id, xNode, yNode));
```

Für die Parameter der Kanten wird gleichermaßen über die Liste der „fileEdges “ iteriert.

```
84          for(String edgeString : fileEdges){
85              String[] edgesParams = edgeString.split(" ");
86              int idStart = Integer.parseInt(edgesParams[0]);
87              int idTarget = Integer.parseInt(edgesParams[1]);
```

Als Nächstes wird geschaut, welcher Zielknoten gegebenenfalls zu einem Startknoten gehört, um die Knotenpaare einer Kante bestimmen zu können. Hierfür iteriert man durch die Liste der „nodes “ und prüft ab, ob die ID dieser Node mit der ID übereinstimmt, die zuvor in der Datei edges.txt ausgelesen und in „idStart “ bzw. „idTarget “ gespeichert wurde. Wurde ein solcher Knoten gefunden, dann wird dieser als Startknoten bzw. Endknoten gespeichert, da er ja der Knoten einer Kante ist.

```
92              for(Node node : nodes){
93
94                  if(node.getId() == idStart){
95
96                      startNode = node;
97                      break;
98
99                  }
100
101             }
102
103             Node endNode = null;
104
105             for(Node node : nodes){
106
107                 if(node.getId() == idTarget){
108
109                     endNode = node;
110                     break;
111
112                 }
113
114             }
```

Diese neu zugewiesenen Knoten werden als Parameter benutzt, um eine Kante zu erzeugen. Diese Kante wird zweifach initialisiert, wobei jeweils der Start- und Endknoten einer Kante vertauscht sind, da man in der Klasse Map noch einen ungerichteten Graphen erzeugt. Als letzten Schritt fügt man genau diese Kante der Liste der Kanten des entsprechenden Knotens hinzu.

```
116             Edge startEdge = new Edge(endNode, startNode.distance(endNode));
117             startNode.getEdges().add(startEdge);
118
119             Edge endEdge = new Edge(startNode, endNode.distance(startNode));
120             endNode.getEdges().add(endEdge);
121         }
```

Außerdem soll die Methode draw implementiert werden, so dass diese die Karte zeichnet. Dazu wird mittels einer verschachteteln for-Schleife über die Knoten der Kanten iteriert und es wird dabei jeweils der Anfangs- sowie der Endknoten einer gemeinsamen Kante bestimmt. Diese Kante wird mit der vor implementierten draw()-Methode in schwarz gezeichnet.

```
188     /**
189      * Zeichnen der Karte.
190      */
191     void draw() {
192         // Zeichnet hier alle Kanten der Karte. Hierzu sollte Node.draw benutzt werden.
193         // Es können die Original-Koordinaten aus den Knoten benutzt werden. Diese werden
194         // automatisch geeignet skaliert.
195         for(int m = 0; m < nodes.size(); m++){
196             for (int n = 0; n < nodes.get(m).getEdges().size(); n++){
197                 Node start = nodes.get(m);
198                 Node target = start.getEdges().get(n).getTarget();
199 
200                 start.draw(target, Color.BLACK);
201             }
202         }
203     }
```

Aufgabe 2.2 Positionen wählen

Es soll eine Methode getClosest() implementiert werden, die den dichtesten Knoten zu einer gegebenen Position zurückgibt. Hierbei werden die x- und y-Koordinaten übergeben. Diese Koordinaten werden benutzt, um einen Hilfsknoten „position “ zu erzeugen. Außerdem werden die Hilfsvariablen minDistance und closestNode erstellt. Dann wird über die Liste der Nodes iteriert und es wird bei jedem Durchlauf die Distanz von dem aktuellen Knoten zu der übergebenen Position berechnet, wobei der Knoten mit der geringsten Distanz am Ende in „closestNode “ gespeichert und zurückgegeben wird.

```
205     /**
206      * Findet den dichtesten Knotens zu einer gegebenen Position.
207      *
208      * @param x Die x-Koordinate.
209      * @param y Die y-Koordinate.
210      * @return Der Knoten, der der Position am nächsten ist. null,
211      * falls es einen solchen nicht gibt.
212      */
213     Node getClosest(final double x, final double y) {
214 
215 
216         Node position = new Node(-1, x,y);
217 
218         double minDistance = Double.POSITIVE_INFINITY;
219         Node closestNode= null;
220 
221 
222 
223         for(Node node : nodes) {                        // Iteriert durch Liste mit allen
                Knoten
224             double closest = node.distance(position);
225 
226             if(closest < minDistance) {
227                 minDistance = closest;
228                 closestNode = node;
229             }
230 
231 
232         }
233 
234         return closestNode; // Ersetzen
235     }
```

Aufgabe 2.3 Routenplanung

Es soll die Methode shortestPath() imolementiert werden, die nach dem Dijkstra-Algorithmus den kürzesten Weg zwischen 2 Punkten bestimmt. Die durchsuchten Kanten werden in der Karte blau eingefärbt, während der kürzeste gefundene Weg rot eingefärbt wird.

Zunächst werden die lokalen Variablen ßtartünd ënd"deklariert, die die übergeben Quell- und Zielknoten abspeichern. Im Anschluss erstellt man eine PriorityQueue namens "borderßum Abspeichern des Randes und eine ArrayList namens "chosenßum Abspeichern der Knoten, die zum kürzesten Weg gehören. border hat einen Comperator, der die Liste automatisch erstmal nach dem Knoten mit den kleinsten Kosten sortiert und im Anschluss nach der Id.

```
235     /**
236      * Methode bestimmt den kürzesten Weg zwischen Quell- und Zielknoten.
237      * Zeichnet Rand und kürzesten Weg in die Karte ein.
238      *
239      * @param from Der Quellknoten.
240      * @param to Der Zielknoten.
241      */
242     private void shortestPath(final Node from, final Node to) {
243         Node start = from;
244         Node end = to;
245 
246         Queue<Node> border = new PriorityQueue<>(Comparator.comparingDouble(Node::
              getCosts).thenComparingInt(Node::getId));
247         ArrayList<Node> chosen = new ArrayList();
```

Nun erstellt man eine Kante, die auf sich selber zeigt und die Kosten der Distanz von sich zu sich selber beinhaltet.

```
249         start.reachedFromAtCosts(start, start.distance(start));
```

Nun beginnt eine while-Schleife, die so lange läuft bis der Zielknoten ein Teil von der ArrayList chosen ist, also der Knoten ausgewählt ist und somit der schnellste Weg gefunden ist.

Zunächst wird in die List edges die ausgehenden Kanten dieses Knotens hinzugefügt und der Startknoten wird ausgewählt. In einer inneren for-Schleife geht man nun die List edges durch und betrachtet vorerst alle Kanten, deren Zielknoten der Kante ein Teil vom Rand sind. Ist sie im Rand, so wird verglichen, ob die Kosten der Kante größer der Summe von der Distanz des Vorgängers zum Quellknoten summiert mit der Distanz des Vorgängers und dem jetzigen Knoten ist. Trifft dies zu, so wird die Kante entfernt und mit dem neuen Weg, der kürzer ist, überschrieben und wieder eingefügt. Ist der Knoten nicht im Rand und auch nicht ausgewählt, so wird eine Kante erzeugt, die dem Rand hinzugefügt wird. In beiden Fällen wird dann in die Karte die Kante als blaue Linie eingezeichnet.

Am Ende wird nochmal der Knoten mit dem kleinsten Kosten bestimmt und ausgewählt, sodass die Schleife dies nochmal von vorne laufen lassen kann.

```
251         while (!chosen.contains(end)) {
252 
253             List<Edge> edges = start.getEdges();
254             chosen.add(start);
255 
256             for (Edge edge : edges) {
257                 if (border.contains(edge.getTarget())) {
258                     if (edge.getTarget().getCosts() > (start.getCosts() + edge.getCosts()
                          )) {
259                         border.remove(edge.getTarget());
260                         edge.getTarget().reachedFromAtCosts(start, start.getCosts() +
                              edge.getCosts());
261                         border.add(edge.getTarget());
262                         start.draw(edge.getTarget(), Color.blue);
263                     }
264                 } else if (!chosen.contains(edge.getTarget())) {
```

```
265                     edge.getTarget().reachedFromAtCosts(start, start.getCosts() + edge.
                          getCosts());
266                     border.add(edge.getTarget());
267                     start.draw(edge.getTarget(), Color.blue);
268                 }
269             }
270             start = border.poll();
271             chosen.add(start);
272 
273         }
```

Letzendlich geht man rekursiv den kürzesten Weg zurück und zeichnet diesen rot in die Karte ein, solange der Vorgänger auf dem Weg vom Startknoten zu diesem Knoten nicht der Endknoten selber ist.

```
275         if (end.getFrom() != null) {
276             Node chosenL = end;
277             while (chosenL.getFrom() != chosenL) {
278                 chosenL.draw(chosenL.getFrom(), Color.red);
279                 chosenL = chosenL.getFrom();
280             }
281         }
```