

Lösungsvorschlag

```

20 */
21 List<Node> nodes = new ArrayList<>();
22
23 /**
24  * Konstruktor. Liest die Karte ein.
25  *
26  * @throws FileNotFoundException Entweder die Datei "nodes.txt" oder die
27  *                               Datei "edges.txt" wurden nicht gefunden.
28  * @throws IOException           Ein Lesefehler ist aufgetreten.
29  */
30 Map() throws FileNotFoundException, IOException {
31 List<String> fileEdges = new ArrayList<>();
32 List<String> fileNodes = new ArrayList<>();
33
34 try (BufferedReader stream = new BufferedReader(new InputStreamReader(new FileInputStream(
35     "edges.txt")))) {
36 String line;
37 while ((line = stream.readLine()) != null) {
38 fileEdges.add(line);
39 }
40 } catch (FileNotFoundException e) {
41 throw new IllegalArgumentException("'edges.txt' wurde nicht gefunden.");
42 } catch (IOException e) {
43 throw new IllegalArgumentException("Ein Lesefehler ist aufgetreten.");
44 }
45 try (BufferedReader stream = new BufferedReader(new InputStreamReader(new FileInputStream(
46     "nodes.txt")))) {
47 String line;
48 while ((line = stream.readLine()) != null) {
49 fileNodes.add(line);
50 }
51 } catch (FileNotFoundException e) {
52 throw new IllegalArgumentException("'nodes.txt' wurde nicht gefunden.");
53 } catch (IOException e) {
54 throw new IllegalArgumentException("Ein Lesefehler ist aufgetreten.");
55 }
56 for (String nodeLine : fileNodes) {
57 String[] nodeParams = nodeLine.split(" ");
58 int id = Integer.parseInt(nodeParams[0]);
59 double xNode = Double.parseDouble(nodeParams[1]);
60 double yNode = Double.parseDouble(nodeParams[2]);
61 nodes.add(new Node(id, xNode, yNode));
62 }
63
64 for (String edgeString : fileEdges) {
65 String[] edgesParams = edgeString.split(" ");
66 int idStart = Integer.parseInt(edgesParams[0]);
67 int idTarget = Integer.parseInt(edgesParams[1]);
68
69 Node startNode = null, endNode = null;
70
71 for (Node node : nodes) {
72 if (node.getId() == idStart) {
73 startNode = node;
74 } else if (node.getId() == idTarget) {
75 endNode = node;
76 }
77 }
78
79 if (startNode != null && endNode != null) {
80 startNode.getEdges().add(new Edge(endNode, startNode.distance(endNode)));
81 endNode.getEdges().add(new Edge(startNode, endNode.distance(startNode)));
82 }
83 }
84 }
85
86 /**
87  * Zeichnen der Karte.
88  */
89 void draw() {
90 for (Node node : nodes) {

```

```

91 for (Edge edge : node.getEdges()) {
92     node.draw(edge.getTarget(), Color.BLACK);
93 }
94 }
95 }

```

Aufgabe 2.2 Positionen wählen

Es soll eine Methode `getClosest()` implementiert werden, die den nächstgelegenen Knoten zu einer gegebenen Position zurückgibt. Die Methode erhält als Parameter die x- und y-Koordinaten der Position. Zunächst wird ein Hilfsknoten namens `position` mit den übergebenen Koordinaten erstellt. Zudem werden Hilfsvariablen `minDistance` und `closestNode` initialisiert. Anschließend wird über die Liste der Knoten (`Nodes`) iteriert. In jeder Iteration wird die Distanz von dem aktuellen Knoten zu der übergebenen Position berechnet. Der Knoten mit der geringsten Distanz wird in der Variable `closestNode` gespeichert. Am Ende wird dieser Knoten zurückgegeben.

```

97 /**
98  * Findet den dichtesten Knoten zu einer gegebenen Position.
99  *
100  * @param x Die x-Koordinate.
101  * @param y Die y-Koordinate.
102  * @return Der Knoten, der der Position am nächsten ist. null,
103  * falls es einen solchen nicht gibt.
104  */
105 Node getClosest(final double x, final double y) {
106     Node position = new Node(-1, x, y);
107     double minDistance = Double.POSITIVE_INFINITY;
108     Node closestNode = null;
109
110     for (Node node : nodes) {
111         double distance = node.distance(position);
112         if (distance < minDistance) {
113             minDistance = distance;
114             closestNode = node;
115         }
116     }
117
118     return closestNode;
119 }

```

Aufgabe 2.3 Routenplanung

Implementiert die `shortestPath`-Methode der `RoutePlanner`-Klasse als Suche nach dem kürzesten Weg nach Dijkstra. Zeichnet die gesuchten Kanten, zum Beispiel in blau. Zeichnet den kürzesten Weg in einer anderen Farbe, z. B. rot.

```

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::
247         getCosts).thenComparingInt(Node::getId)));
248     ArrayList<Node> chosen = new ArrayList();
249
250     start.reachedFromAtCosts(start, start.distance(start));

```

Nun beginnt eine while-Schleife, die solange läuft, bis der Zielknoten ein Teil der ArrayList chosen ist, was bedeutet, dass der schnellste Weg gefunden wurde.

```

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                     )) {
259                     border.remove(edge.getTarget());
260                     edge.getTarget().reachedFromAtCosts(start, start.getCosts() +
261                         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.
266                     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 }

```

Abschließend 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 selbst 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     }
282 }

```