Aufgabe 1: Lisa rennt

Teilnahme-Id: 48325

Bearbeiter/-in dieser Aufgabe:
Alexander von Recum

27. April 2019

Inhaltsverzeichnis

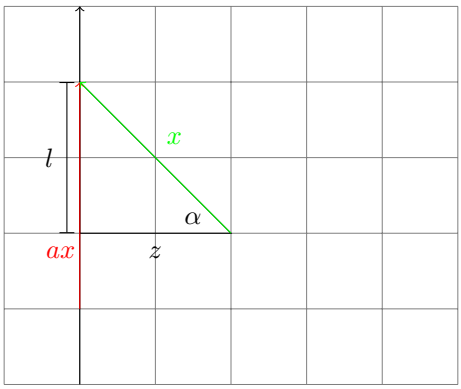
1 Lösungsidee

1.1 Einleitung/Abstract

Wir betrachten das Problem zunächst und versuchen es in seine Unterprobleme zu zerlegen. Zunächst stellt sich die Frage, wie bzw. unter welchem Winkel Lisa rennen würde, wenn es überhaupt keine Hindernisse gäbe, und ob sie immer unter dem gleichen Winkel rennen würde oder z.B. je nach Position unter verschiedenen Winkeln. Wenn dieses Problem gelöst ist, weiß man, wie Lisa sich von allen Punkten, von denen aus sie unter diesem effizientesten Winkel ohne durch Hindernisse durchzugehen zur y-Achse kommen könnte, bewegen würde. Wir bezeichnen diese Punkte als "y-Sichtpunkte", da man von diesen Punkten, wenn man genau in Richtung des effizientesten Wegs schaut, die y-Achse „sehen“ kann. Mit „sehen“ meinen wir, dass eine gegebene Strecke $\overline{PQ}$ (in diesem Falle die Strecke vom y-Sichtpunkt zur y-Achse unter dem effizientesten Winkel) mit allen Strecken, aus denen die einzelnen Polygone bestehen, keinen Schnittpunkt hat. Anschließend muss man überlegen, wie Lisa sich am schnellsten durch die Polygone zu diesen y-Sichtpunkten bewegen kann und herausfinden, welche Punkte y-Sichtpunkte sind. Wenn wir herausgefunden haben, welche Punkte y-Sichtpunkte sind, können wir von ihnen aus Strecken zur y-Achse unter dem errechneten effizientesten Winkel ziehen. Nun muss die minimale Distanz von Lisas Startpunkt zu allen betrachteten Punkten, die auf der y-Achse liegen, berechnet werden. Diese können nun anhanddessen, wie groß die Differenz zwischen der Zeit (in Sekunden), die der Bus braucht, um zu einem y-Schnittpunkt zu kommen und der minimalen Zeit, die Lisa braucht, um zu dem gleichen y-Schnittpunkt zu kommen, verglichen werden. Ist dieser Wert positiv, so kann Lisa nach 7:30 losrennen. Ist er negativ, so muss sie vor 7:30 Uhr losrennen. Der kürzeste Pfad zum y-Schnittpunkt, bei dem dieser Wert am größten ist, ist der Weg, den Lisa nehmen muss.

1.2 Wie würde Lisa ohne Hindernisse rennen?

Wir versuchen die Frage, unter welchem Winkel wir gehen müssen, in ein Extremwertproblem umzuwandeln. Dafür müssen wir eine passende Funktion aufstellen, ihr Maximum finden, das Maximum wieder in die Funktion einsetzen und den Winkel herausfinden. Wir verallgemeinern das Problem, indem wir sagen, der Bus bewegt sich a mal schneller die y-Achse entlang als Lisa sich in der Ebene bewegen kann. In diesem konkreten Falle wäre $a = \frac{30}{15} = 2$. Somit bewegt sich der Bus pro Zeiteinheit mit der Geschwindigkeit ax die y-Achse entlang, während Lisa unter einem bestimmten Winkel α in Richtung y-Achse bewegt. An dieser Stelle ist eine Visualisierung hilfreich.



Wie man hier schon intuitiv erkennen kann, muss $l - ax$ maximiert werden. Je größer diese Differenz, desto mehr Zeit bleibt Lisa, bevor sie losrennen muss, um auf den Bus zu treffen. Um l zu bestimmen, stellen wir $\sin(\alpha) = \frac{l}{x}$ nach l um und erhalten $\sin(\alpha)x = l$. Aus $\alpha = arccos(\frac{z}{x})$ ergibt sich die Gleichung

$$f(x) = \sin(\arccos(\frac{z}{x}))x - ax$$

Unser Ziel ist es nun, den x-Wert eines Maximums zu finden, für das gilt:

$$x \geq z; x, z > 0$$

Wir leiten die Funktion f ab und erhalten:

$$f'(x) = \frac{z^2}{x^2\sqrt{1 - (\frac{z}{x})^2}}$$

Für ein Maximum gilt die notwendige Bedingung $f'(x) = 0$ und die hinreichende Bedingung $f''(x) < 0$. Wir stellen die Funktion nach x um und erhalten die Nullstellen:

$$x_1 = \frac{z}{\sqrt{-a^{-2}+1}}, x_2 = -\frac{z}{\sqrt{-a^{-2}+1}},$$

Da die Wurzel im Nenner und z immer positiv sind, ist x_2 immer negativ. Somit ist x_1 unser gesuchtes Maximum. Die hinreichende Bedingung wird nicht aufgeschrieben, sie ist aber ebenfalls erfüllt. Wir können nun unser x_1 in unsere Formel $\alpha = \arccos(\frac{z}{x})$ einsetzen. Nach Umformungen erhalten wir:

$$\alpha = \arccos(\sqrt{-a^{-2}+1})$$

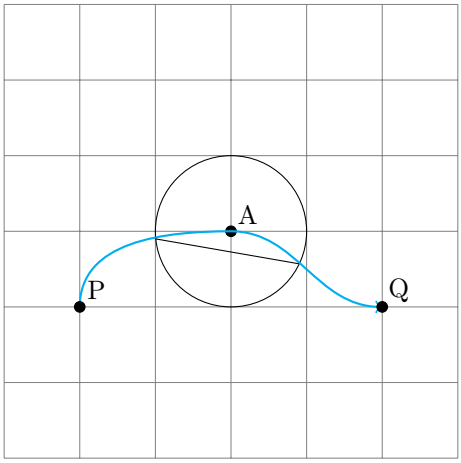
Somit haben wir eine allgemeine Formel für die Berechnung des effizientesten Winkels. Dass der kürzeste Weg zur y-Achse ein gradliniger ist, ist aufgrund der Dreiecksungleichung trivial. Für unseren konkreten Fall setzen wir $a = 2$ ein und erhalten:

$$\alpha = \arccos(\sqrt{-2^{-2}+1}) = \arccos(\sqrt{\frac{1}{4}}) = arccos(\frac{1}{2}) = 30°$$

Lisa muss somit in diesem konkreten Fall immer im Winkel von 30° rennen. Zur Verallgemeinerung des Problems kann die allgemeine Formel genutzt werden.

1.3 Wie bewegt man sich am schnellsten durch die Polygone?

Nun ist bewiesen, dass der Winkel, unter dem man von den y-Sichtpunkten aus gehen muss, immer konstant ist. Man könnte also unter diesem Winkel zur y-Achse gehen - wenn es keine Hindernisse gäbe. Um am effizientesten durch die Hindernisse zu gehen, muss man sich über die Punkte, die die Ecken der Polygone bilden, bewegen. Dies kann durch Widerspruch bewiesen werden. Wenn man annimmt, es gäbe einen Punkt A, der nicht auf einer geraden Linie zwischen zwei beliebigen Punkten der Polygone liegt und über den man gehen müsste, um von einem Punkt P zu einem Punkt Q möglichst schnell zu kommen, so kann man um diesen herum immer einen Kreis bilden, durch den eine Strecke geht, die kürzer als die ursprüngliche Strecke ist und nicht über den Mittelpunkt P des Kreises geht. Anders gesagt: die Dreiecksungleichung gilt auch hier.



Somit wird auch klar, welche Punkte y-Sichtpunkte sein können: dies können nur Punkte sein, die Ecke eines der Polygone sind und von denen man unter dem errechneten effizientesten Winkel die y-Achse sieht. Nun kann ein Algorithmus zur Lösung des Problems entwickelt werden.

1.4 Algorithmus

Wir konstruieren einen ungerichteten gewichteten Sichtbarkeitsgraphen G. Die Menge aller Knoten V des Graphen setzt sich zusammen aus:

1. Der Menge aller Polygoneckpunkte P
2. Der Menge aller y-Schnittpunkte X

3. Lisas Startpunkt S

Das Hauptattribut dieses Sichtbarkeitsgraphen ist, dass nur Kanten von Polygoneckenpaaren (A, B) teil der Menge aller Kanten E des Graphen sind, wenn diese Paare keine der die einzelnen durch die Polygonecken gebildeten Strecken schneiden. Hinzu kommen noch die Kanten von den y-Sichtpunkten zu dem unter dem optimalen Winkel (in diesem Falle 30°) verlaufenden y-Schnittpunkten. Diese dürfen nur Teil von E werden, wenn sie ebenfalls keines der Polygone schneiden. Nachdem der Graph konstruiert ist, kann mithilfe des Algorithmus von Dijkstra ein Shortest Path Tree bestimmt werden, sodass für jeden Knoten v des Graphen bekannt ist, was die minimale Distanz vom Startknoten s zu ihm ist und welche Knoten man für diesen Pfad in welcher Reihenfolge abgehen muss.

1.4.1 Konstruktion des Sichtbarkeitsgraphen

Es wird angenommen, dass alle Strecken beim Einlesen in der Menge aller Polygonstrecken L gespeichert wurden. Um den Sichtbarkeitsgraphen zu konstruieren, gehen wir die Menge aller Polygoneckpunkte $P \cup S$ durch. Für jeden der Eckpunkte gehen wir erneut die Menge $P \cup S$ durch und verbinden dabei den aktuellen Punkt I mit allen Punkten J (ohne dabei den Knoten mit sich selbst zu verbinden). Nun gehen wir für jede dieser Verbindungen die Menge L durch. Sobald die Strecke $\overline{IJ}$ eine Strecke in der Menge L schneidet, verwerfen wir diese Kante. Falls sie nach überprüfen aller Elemente aus L keine der Strecken schneidet, fügen wir sie dem Graphen hinzu. Bevor alle Elemente aus L überprüft werden muss bei der Kante noch sichergestellt werden, ob sie nicht innerhalb eines Polygons verläuft und so ungültig ist. Dies kann nur der Fall sein, wenn die Polygon-ID von I mit der von J übereinstimmt. Dafür kann der Ray Casting Algorithm genutzt werden. Dafür wird der Mittelpunkt zwischen I und J bestimmt. Man schießt in eine beliebige Richtung einen Strahl (z.B. parallel zur x-Achse nach rechts) und zählt, wie viele Schnittpunkte der Strahl mit dem Polygon hat. Dafür müssen alle Kanten des Polygons überprüft werden. Ist die Anzahl der Schnittpunkte ungerade, so liegt die Kante innerhalb des Polygons und kann direkt verworfen werden. Ansonsten wird die obengenannte Überprüfung begonnen. Beim Ray Casting Algorithm muss eine Ausnahme beachtet werden. Stimmt die y-Koordinate einer Polygonstrecke mit der y-Koordinate der nächsten Strecke und der y-Koordinate des Strahls überein (falls wir ihn parallel zur x-Achse nach rechts schießen), so muss dieser Schnitt nur einmal gezählt werden, ansonsten liefert der Algorithmus fehlerhafte Resultate. Details zu diesem Spezialfall sind in der Implementierung und in den Beispielen erwähnt. Nachdem man diese Kanten der Menge E hinzugefügt hat, müssen noch die Kanten von den y-Sichtpunkten zu den y-Schnittpunkten überprüft werden. Der Algorithmus hier ist derselbe, wie zuvor. Man muss davor noch die y-Koordinate des y-Schnittpunkts ausgehend von der x-Koordinate des y-Sichtpunkts und dem Winkel ausrechnen. Die x-Koordinate des y-Schnittpunkts ist immer 0. Sobald man den Graphen konstruiert hat, kann man zum nächsten Schritt übergehen: dem Finden des kürzesten Weges.

1.4.2 Algorithmus von Dijkstra

Der Algorithmus von Dijkstra zum Finden eines Shortest Path Tree kann in 3 Schritte unterteilt werden:

1. Man erstelle eine Menge aller besuchten Knoten M
2. Man initialisiere für alle Knoten V des Graphen G die Distanz zum Knoten S als unendlich, außer für den Knoten S selbst. Dieser hat eine Distanz von 0.
3. Solange M weniger Elemente als V hat: Man finde vom aktuellen Knoten v einen noch nicht in M enthaltenen Knoten u, der von v die geringste Distanz hat. Man füge u zu M hinzu. Man erneuere alle minimalen Distanzen und Pfade von allen benachbarten Knoten von u, falls die Distanz bzw. der Pfad von S zu u und die Distanz bzw. der Pfad von u zu diesem benachbarten Knoten kürzer ist, als die aktuelle Distanz bzw. der Pfad. Man gehe zu Schritt 3.

1.4.3 Bestimmen des optimalen Weges

Nachdem man für jede Kante die minimale Distanz und den Pfad berechnet hat, geht man noch einmal alle Knoten V durch. Die Knoten K, die eine x-Koordinate von 0 haben (also nicht nur y-berechnete Schnittpunkte, sondern auch Polygoneckpunkte mit einer x-Koordinate von 0) werden nach folgendem Kriterium verglichen: Man zieht die Zeit, die Lisa braucht, um zu einem Knoten K zu laufen von der Zeit ab, die der Bus braucht, um die y-Koordinate des Knotens K zu erreichen, ab. So:

$$yKoordinateVonKinMetern/(GeschwindigkeitBus/3,6) - LisaWegZuKinMetern/(GeschwindigkeitLisa/3,6)$$

Wir suchen durch eine lineare Suche den Knoten K, wo dieser Wert am größten ist. Sobald wir alle Elemente durchgegangen sind, haben wir ihn gefunden. Wir Addieren diese Zeit (in Sekunden) auf 7:30 auf (falls die Zahl negativ ist, muss Lisa vor 7:30 Uhr losgehen) und haben so die kürzeste Uhrzeit und den kürzesten Weg.

1.5 Laufzeit- und Speicheranalyse

Die Erstellung des Sichtbarkeitsgraphen hat im schlimmsten Fall eine Zeitkomplexität von $O(n^3)$. Der Algorithmus von Dijkstra hat eine Zeitkomplexität von $O(n^2)$. Das Finden des Knotens mit dem kürzesten Wegs hat eine Zeitkomplexität von $O(n)$. Somit läuft der ganze Algorithmus in $O(n^3)$. Aufgrund der Adjazenzmatrix des Graphen hat der Algorithmus eine Speicherkomplexität von $O(n^2)$. Alle anderen Speichernutzungen haben eine Komplexität von $O(n)$ (z.B. zum Speichern aller Knoten des Graphen) bzw. $O(1)$ (z.B. zum Speichern des Startpunkts).

2 Umsetzung

Die Implementierung besteht aus folgenden Klassen:

1. der Klasse Main
2. der Klasse WeightedUndirectedGraph
3. der Klasse Set
4. der Klasse LineSegment
5. der Klasse Obstacle
6. der Klasse SVGOutputter
7. der Klasse Static

2.1 Die Klasse Main

Diese Klasse ist für die Lösung verantwortlich. Sie enthält die folgenden Methoden:

1. static void main(String[] args) zum Lösen der Aufgabe
2. static double computeAngle(double geschwindigkeitMaedchen, double geschwindigkeitBus) zum Errechnen des optimalen Winkels für gegebene Geschwindigkeiten von Lisa und Bus
3. static double computeYShift(double xDistance) für die Errechnung vom benötigten δy für ein vorgegebenes x, um unter dem optimalen Winkel zur y-Achse zu kommen

Die Klasse hat die Attribute:

1. static Point startingPoint zur Speicherung des Startpunktes S
2. static Obstacle[] obstacles zur Speicherung der Hindernisse
3. static double angle zur Speicherung des optimalen Winkels

Zur Lösung werden zuerst die Kommandozeilenargumente gelesen. Falls 4 Argumente vorhanden sind, so wird versucht, die neuen Geschwindigkeiten von Lisa und Bus zu lesen. Anschließend wird der optimale Winkel errechnet und in angle gespeichert. Danach wird ein Graph mit der Knotenanzahl 10 erstellt. Der Graph vergrößert sich automatisch, sobald er seine maximale Kapazität erreicht hat, dies wird genauer im nächsten Abschnitt erklärt. Dem Graphen werden dann die Geschwindigkeit von Mädchen und Bus übergeben, damit er mit ihnen Berechnungen ausführen kann. Nun werden die Daten mithilfe eines BufferedReader eingelesen, der die Datei in dem als Kommandozeilenargument übergebenen Pfad zu öffnen versucht. Dabei kann eine FileNotFoundException auftreten, deshalb ist dieser Block von einem try-catch Block umgeben. Als nächstes wird die Anzahl von Hindernissen gelesen und in amountOfObstacles gespeichert. Eine Schleife wird begonnen, die nacheinander die Hindernisse erstellt und ihre Punkte einliest und verbindet. Dafür wird jeweils eine Zeile gelesen, das 0-te Element im String array l ist die Anzahl der Punkte des Graphen. Ein neues Hindernis wird erstellt und im obstacles-Array gespeichert.

dann werden nacheinander für alle Punkte Objekte der Klasse LineSegment vom aktuellen Punkt zum nächsten Punkt erstellt und dem Obstacle-Objekt hinzugefügt. Zum Schluss wird noch der Startpunkt eingelesen. Jetzt wird der Sichtbarkeitsgraph gebaut. Dafür wird der in Punkt 1 erwähnte Algorithmus angewendet. Die genaue Funktion jeder Zeile ist im Code mit Kommentaren erklärt. Anschließend wird der Sichtbarkeitsalgorithmus noch für mögliche y-Schnittpunkte und ihre jeweiligen y-Sichtpunkte ausgeführt. Wenn die Strecke kein Polygon schneidet, wird dem Graphen noch der y-Schnittpunkt und die Kante zu ihm hinzugefügt. Dann wird die Methode dijkstra des Graphen mit dem index des Startpunkts aufgerufen: Der Dijkstra-Algorithmus wird gestartet. Anschließend wird der optimale Pfad für Lisa gefunden und in der ArrayList shortestPath gespeichert. Diese enthält die Indizes der Knoten, die Lisa besuchen muss, um möglichst kurz zu laufen. Dann werden die Ergebnisse dem Graphen entnommen und ausgegeben. Zum Schluss wird noch der DOM-Tree für die SVG mit der Klasse SVGOutputter gebaut und die SVG-Datei im als 2. Kommandozeilenargument übergebenen Pfad gespeichert.

2.2 Die Klasse WeightedUndirectedGraph

Diese Klasse ist eine eigene Implementierung eines Graphen mithilfe von einer Adjazenzmatrix, die zur Verallgemeinerung einige Methoden enthält, die in der Lösung nicht genutzt werden, aber trotzdem für einen Graphen notwendig sind. Sie enthält folgende Methoden:

1. Den Konstruktor WeightedUndirectedGraph(int size), der einen Graph mit der als Argument übergebenen Kapazität erstellt. Die Kapazität wird, falls sie durch Aufruf von addNode zur Hälfte ausgeschöpft ist, verdoppelt.
2. Die Methode connect(int startNode, int endNode, int size), die ein neues Edge-Objekt erstellt und dieses in der Adjazenzmatrix für [startNode][endNode] und [endNode][startNode] speichert.
3. Die Methode disconnect(int startNode, int endNode), die die Verbindung zwischen 2 Kanten mit den jeweiligen Indizes löscht
4. Die Methode isConnected, zur Überprüfung, ob ein Knoten mit einem anderen verbunden ist
5. Die Methode getWeight(int startNode, int endNode), die das Gewicht zweier verbundener Knoten zurückgibt
6. Die Methode addNode(Point a, int oIndex), die dem Graphen einen Knoten hinzufügt und den index des Knotens zurückgibt. Falls die Hälfte der aktuellen maximalen Kapazität des Graphen erreicht ist, wird die Adjazenzmatrix und die Liste der Knoten um das doppelte vergrößert und kopiert. Diese Methode läuft im schlimmsten Fall in $O(n^2)$, allerdings wird die Methode nur jedes 10×2^n-te Mal aufgerufen, weshalb sie in amortisierter linearer Zeit läuft.
7. Die Methode findUnvisitedNeighbor(int index), die für einen gegebenen Knoten einen noch nicht besuchten Nachbarknoten findet. Sie läuft im schlimmsten Falle in $O(n)$.
8. Die Methode dijkstra(int startIndex), die den in Abschnitt 1 beschriebenen Dijkstra-Algorithmus implementiert.
9. Die Methode resetNodes, die alle Knoten des Graphen zurücksetzt.
10. getter-Methoden, die die Ergebnisse des Dijkstra-Algorithmus zurückgeben
11. die Methode int size(), die die Anzahl der Elemente im Graphen zurückgibt

Und folgende Klassen:

1. die Klasse Node, die einen Knoten modelliert
2. die Klasse Edge, die eine Kante modelliert

Und folgende Attribute:

1. private int CURRENT_CAPACITY, die aktuelle Kapazität des Graphen
2. Node[] vertices, die Knoten
3. Edge[][] adjMat, die Kanten

4. private int size, den letzen Index eines Knotens in vertices
5. double geschwindigkeitMaedchen, Lisas Geschwindigkeit im km/h
6. double geschwindigkeitBus, die Geschwindigkeit des Busses im km/h
7. private double latestLeavingTime, die letze Zeit, zu der Lisa das Haus in Sekunden verlassen muss
8. private double meetingTime, die Zeit, zu der Lisa auf den Bus trifft
9. private double meetingYCoord, die y-Koordinate, an der Lisa auf den Bus trifft
10. private int shortestPathNodeIndex, der Index des Knotens im Array vertices, zu dem Lisa gehen muss.

2.3 Die Klasse Set

Diese Klasse wird für den Dijkstra-Algorithmus benötigt um zu überprüfen, ob ein Knoten bereits in der Menge M ist. Sie speichert einen boolean-Array zu den Indizes, der beschreibt, welche Knoten in der Menge sind.

2.4 Die Klasse LineSegment

Diese Klasse modelliert eine Strecke. Sie enthält eine Methode zur Überprüfung, ob sie einen Schnittpunkt mit einer anderen Strecke hat sowie eine Methode, die true zurückgibt, wenn die ihr übergebene Strecke mit ihr gleich ist.

2.5 Die Klasse Obstacle

Diese Klasse modelliert ein Hindernispolygon. Die Anzahl der Punkte des Polygons muss beim Erstellen des Objekts bekannt sein. Sie hat folgende Methoden:

1. die Methode addLineSegment(LineSegment ls), die die übergebene Strecke im Array lineSegments speichert, falls die Kapazität des Hindernisses noch nicht ausgeschöpft ist
2. die Methode intersects(LineSegment l), die eine übergebene Strecke darauf überprüft, ob sie das aktuelle Hindernis schneidet. Sie ruft die methode intersects der übeggebenen Strecke l für alle Strecken des Hindernisses auf. Sobald diese das erste Mal true zurückgibt, gibt diese Methode auch true zurück. Falls die Strecke keine der Strecken schneidet, gibt die Methode false zurück.
3. die Methode isInsideObstacle(LineSegment l), die für eine übergebene Strecke überprüft, ob diese innerhalb des aktuellen Hindernisses liegt, oder nicht. Falls das der Fall ist, gibt sie true zurück, sonst false. Sie benutzt den Ray Casting Algorithmus, um zu überprüfen, ob eine Strecke im Polygon liegt. Der Mittelpunkt zwischen Start- und Endpunkt der Strecke wird errechnet und falls dieser innerhalb des Polygons liegt, so liegt die Strecke auch innerhalb des Polygons. Dafür wird ein Strahl entlang der y-Achse nach rechts geschossen und gezählt, wie oft dieser die Strecken des Polygons schneidet. Ist diese Zahl ungerade, so liegt der Mittelpunkt im Polygon, und die Strecke folglich auch.
4. die Methode countIntersections(LineSegment ray, LineSegment ls), die für einen Strahl und eine Strecke überprüft, wie oft der Strahl vom Mittelpunkt der Strecke die Hindernisse des Polygons schneidet. Hier wird ein Trick verwendet: Falls die Methode eine Strecke, die mit l übereinstimmt, findet, wird -1 zurückgegeben. So werden Strecken, die Teil des Hindernisses sind, auch verbunden. Ein weiterer Trick wird beim Zählen der Schnittpunkte benutzt: Es kann sein, dass der Strahl zwei Strecken gleichzeitig schneidet. Um doppeltes Zählen eines Schnittpunkts zu vermeiden, wird überprüft, ob die vorige Strecke bereits den Strahl geschnitten hat. Falls dies der Fall war, wird der aktuelle Schnitt nicht mitgezählt. Dies liefert zwar falsche Resultate in manchen Spezialfällen, aber die zurückgegebene Zahl ist trotzdem immer dann ungerade, wenn der überprüfte Punkt im Hindernis liegt.
5. die Methode rayTouchesLineSegment(LineSegment ls), die von countIntersections genutzt wird, um doppeltes Schneiden zu vermeiden

2.6 Die Klasse SVGOutputter

Diese Klasse wurde unten nicht aufgeführt, da ihre Implementierung für die Aufgabe nicht relevant ist. Sie erstellt einen DOM-Tree mit allen Hindernissen, dem Startpunkt und dem optimalen Weg. Dieser wird dann in eine SVG-Datei geschrieben. Für das Erstellen des DOM-Trees wird die externe Bibliothek org.w3c.dom bzw. w3c-dom.jar genutzt, für das Schreiben der SVG die Bibliothek org.apache.batik bzw batik-all-1.11.jar.

2.7 Die Klasse Static

Diese Klasse enthält die statische Methode double round(double d). Sie rundet die eingegebene Zahl auf so viele Nachkommastellen genau, wie in der variable *digits* definiert ist. Dafür wird die Zahl mit 10^{digits} multipliziert, auf die nächste ganze Zahl gerundet und durch 10^{digits} geteilt.

3 Beispiele

3.1 Beispiel 1

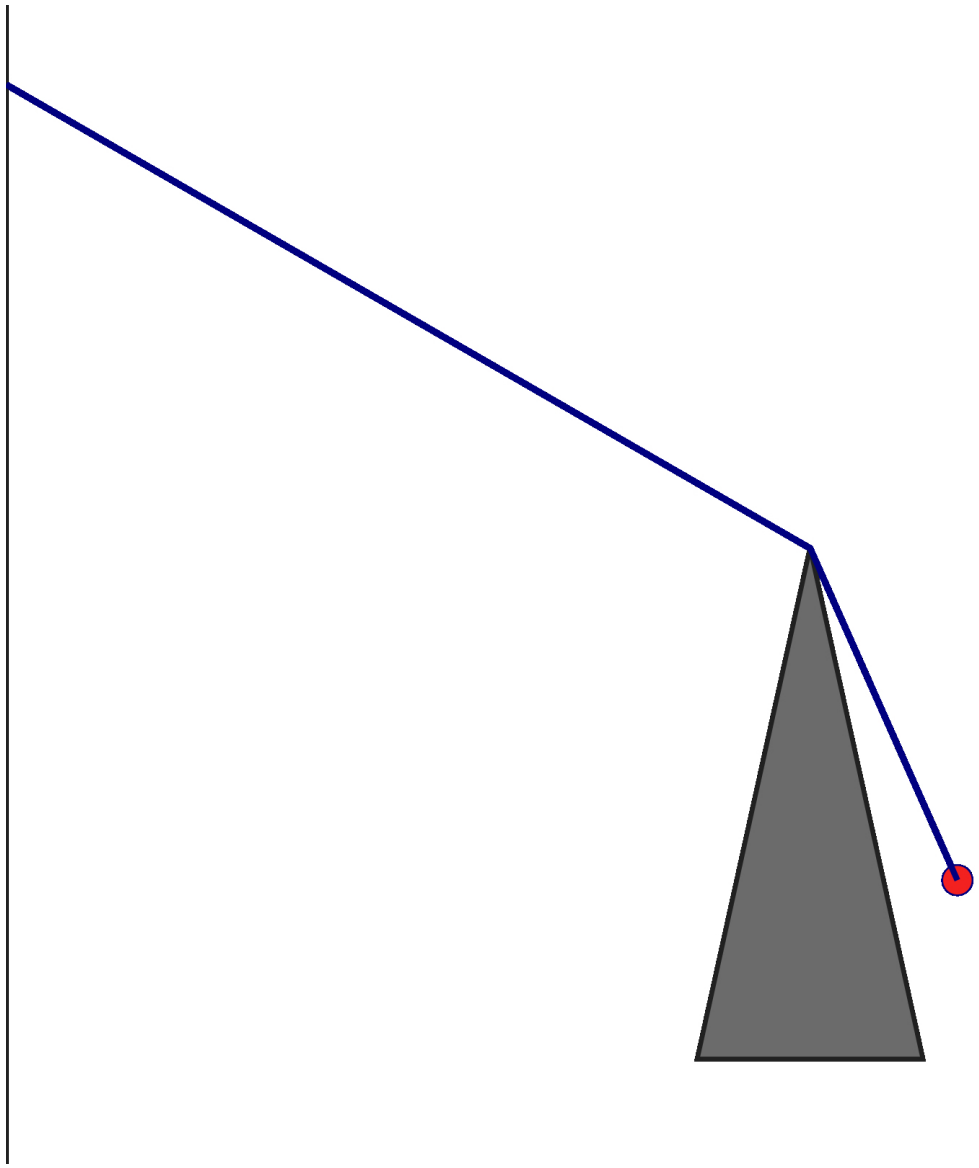
Eingabe

```
1
3 535 410 610 70 460 70
633 189
```

Ausgabe

```
Dijkstra brauchte 1 Millisekunden.
Lisa muss anfangen zu rennen um 07:28:00
Lisa trifft den Bus um 07:31:26
Sie trifft den Bus an der y-Koordinate 718.882
Lisa braucht 00:03:26
Ihr Weg hat eine Laenge von 859.518
Sie geht ueber folgende Knoten:
Knoten (633.0|189.0) ID: L
Knoten (535.0|410.0) ID: P1
Knoten (0.0|718.882) ID: ENDE
```

Visuelle Darstellung



3.2 Beispiel 2

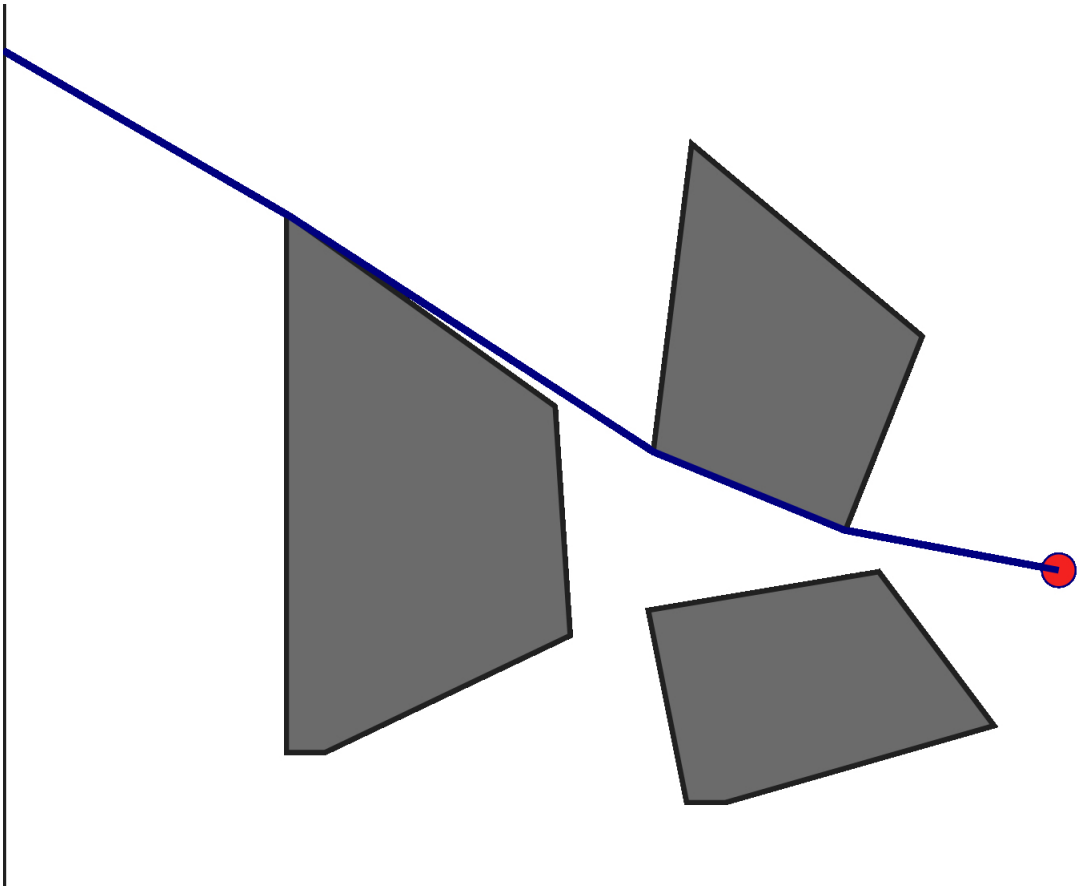
Eingabe

```
3
4 390 260 505 213 551 329 413 444
5 410 50 433 50 594 96 525 188 387 165
5 170 80 193 80 340 150 331 287 170 402
633 189
```

Ausgabe

```
Dijkstra brauchte 1 Millisekunden.
Lisa muss anfangen zu rennen um 07:28:09
Lisa trifft den Bus um 07:31:00
Sie trifft den Bus an der y-Koordinate 500.149
Lisa braucht 00:02:51
Ihr Weg hat eine Laenge von 712.608
Sie geht ueber folgende Knoten:
Knoten (633.0|189.0) ID: L
Knoten (505.0|213.0) ID: P1
Knoten (390.0|260.0) ID: P1
Knoten (170.0|402.0) ID: P3
Knoten (0.0|500.149) ID: ENDE
```

Visuelle Darstellung



3.3 Beispiel 3

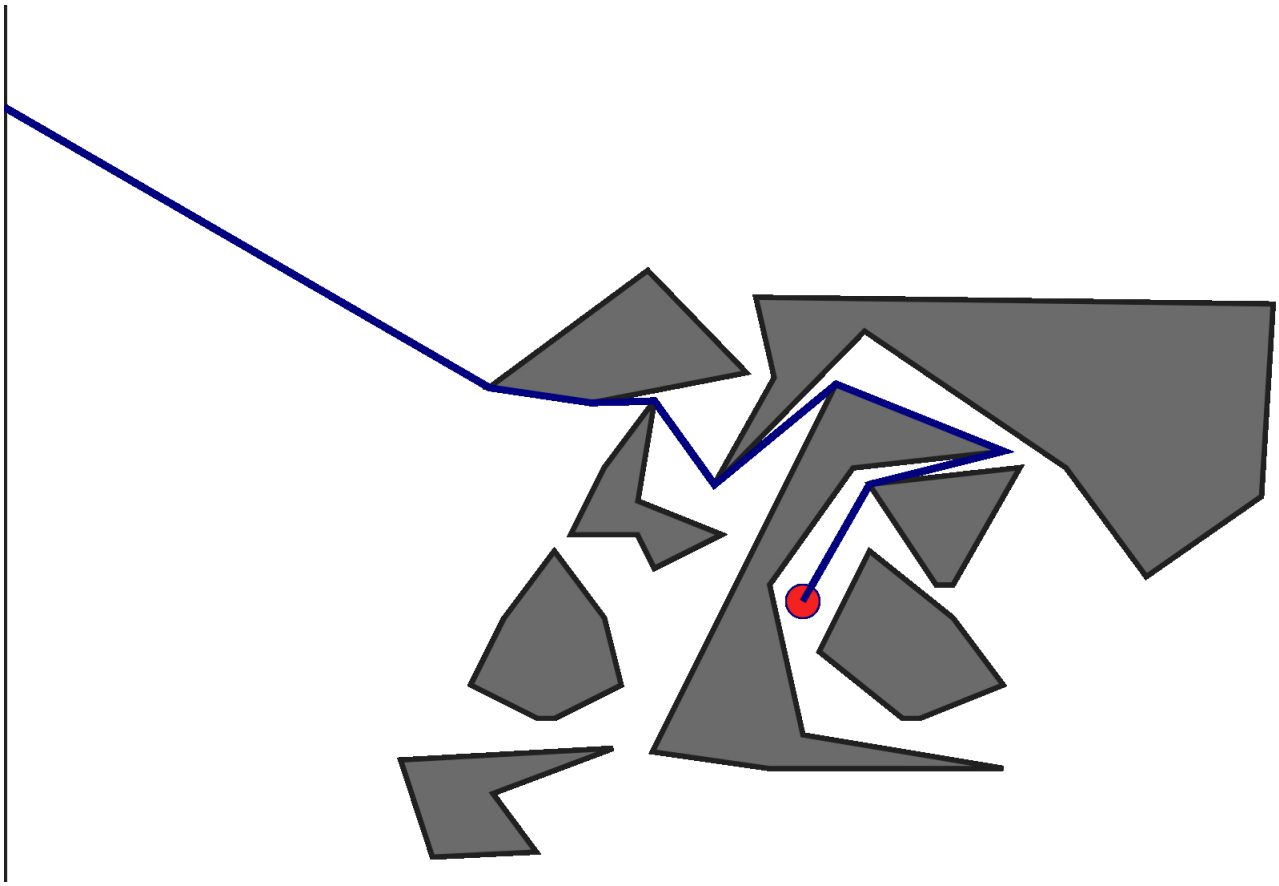
Eingabe

```
8
6 539 98 549 98 599 118 569 158 519 198 489 138
4 559 178 569 178 609 248 519 238
8 389 78 459 68 599 68 479 88 459 178 509 248 599 258 499 298
7 320 98 330 98 370 118 360 158 330 198 300 158 280 118
7 380 208 390 188 430 208 380 228 390 288 360 248 340 208
4 352 287 445 305 386 366 291 296
5 319 18 293 53 365 80 238 73 257 15
9 637 248 516 330 426 238 462 302 451 350 613 348 761 346 754 231 685 183
479 168
```

Ausgabe

```
Dijkstra brauchte 1 Millisekunden.
Lisa muss anfangen zu rennen um 07:27:29
Lisa trifft den Bus um 07:30:56
Sie trifft den Bus an der y-Koordinate 464.008
Lisa braucht 00:03:27
Ihr Weg hat eine Laenge von 862.58
Sie geht ueber folgende Knoten:
Knoten (479.0|168.0) ID: L
Knoten (519.0|238.0) ID: P2
Knoten (599.0|258.0) ID: P3
Knoten (499.0|298.0) ID: P3
Knoten (426.0|238.0) ID: P8
Knoten (390.0|288.0) ID: P5
Knoten (352.0|287.0) ID: P6
Knoten (291.0|296.0) ID: P6
Knoten (0.0|464.008) ID: ENDE
```

Visuelle Darstellung



3.4 Beispiel 4

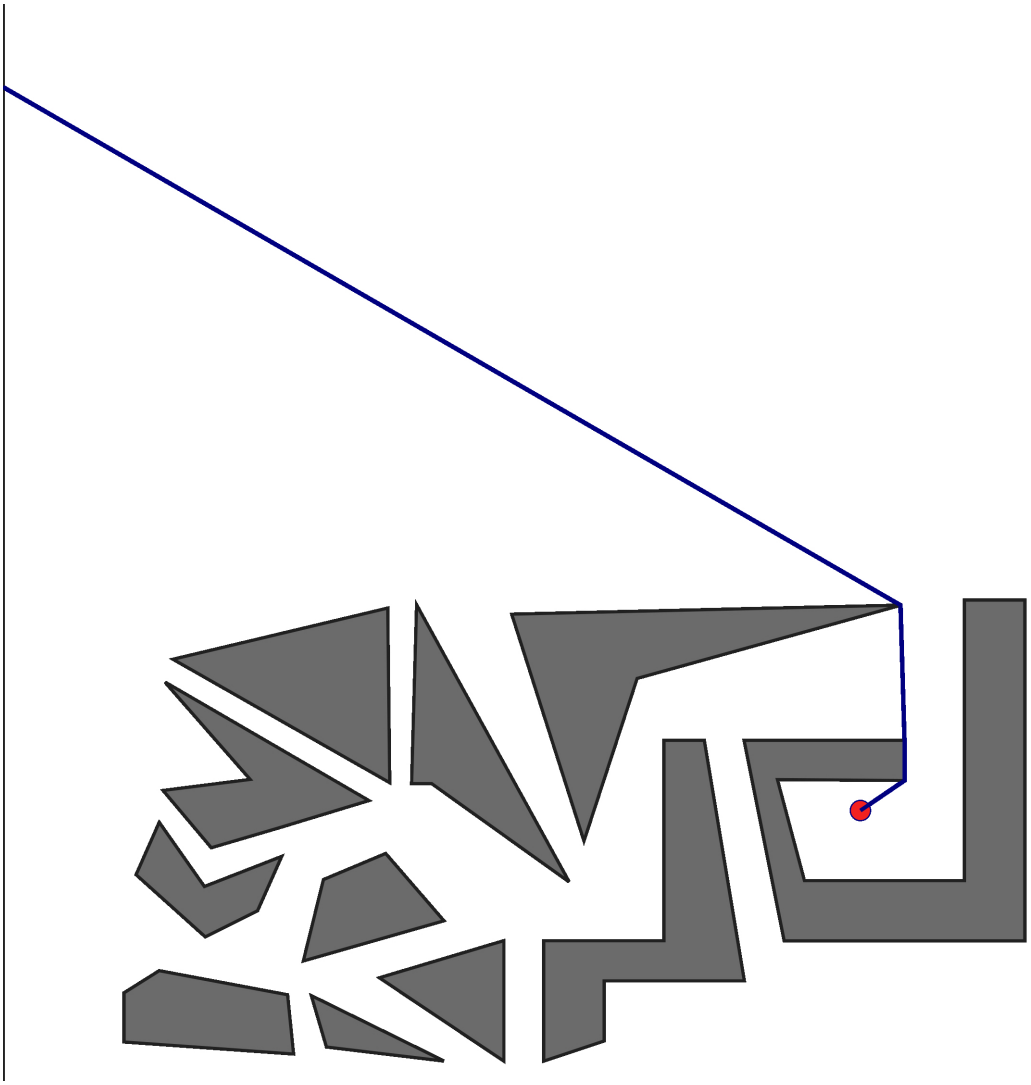
Eingabe

```
11
5 121 39 290 27 284 86 156 110 121 88
6 133 206 202 144 254 170 278 224 201 194 156 258
6 160 290 247 301 162 398 365 280 276 253 208 233
3 170 421 386 298 384 472
4 408 297 428 297 565 199 413 475
4 300 120 440 160 382 227 320 201
3 323 34 440 20 308 85
3 500 20 500 140 376 103
8 540 20 600 40 600 100 740 100 700 340 660 340 660 140 540 140
4 580 240 633 402 896 475 508 466
10 780 140 1020 140 1020 480 960 480 960 200 800 200 773 301 900 300 900 340 740 340
856 270
```

Ausgabe

```
Dijkstra brauchte 2 Millisekunden.
Lisa muss anfangen zu rennen um 07:26:56
Lisa trifft den Bus um 07:31:59
Sie trifft den Bus an der y-Koordinate 992.305
Lisa braucht 00:05:03
Ihr Weg hat eine Laenge von 1262.924
Sie geht ueber folgende Knoten:
Knoten (856.0|270.0) ID: L
Knoten (900.0|300.0) ID: P11
Knoten (900.0|340.0) ID: P11
Knoten (896.0|475.0) ID: P10
Knoten (0.0|992.305) ID: ENDE
```

Visuelle Darstellung



3.5 Beispiel 5

Eingabe

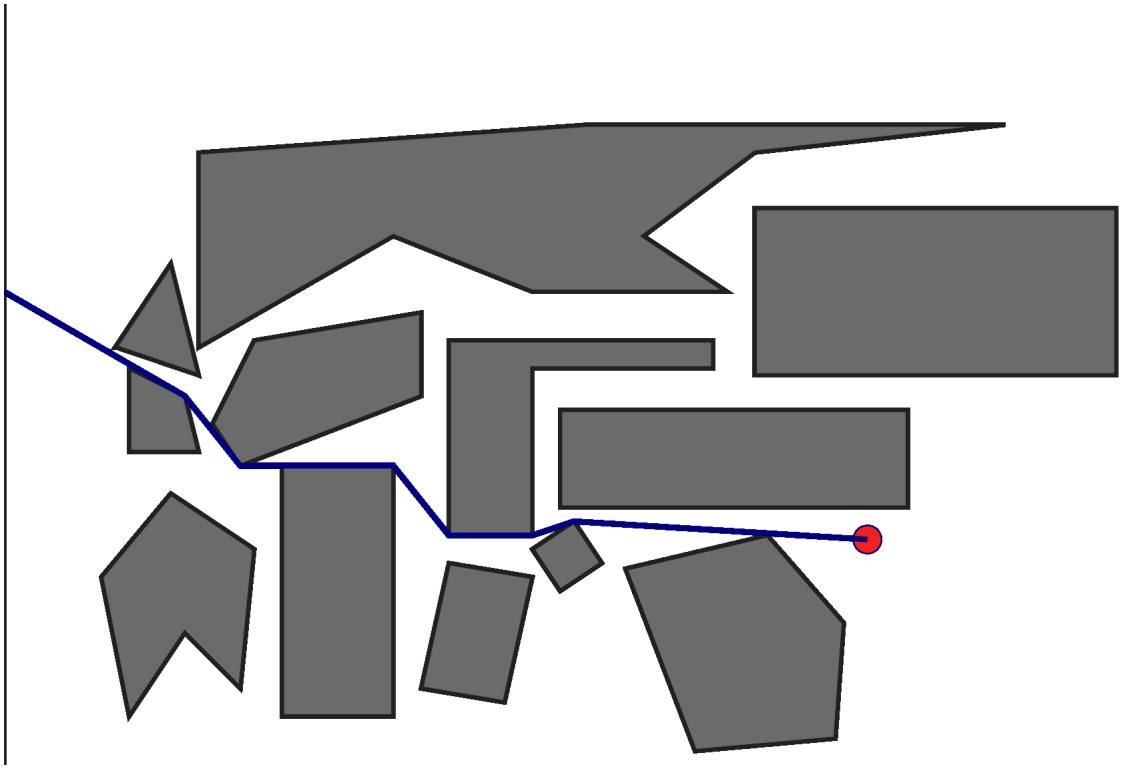
```
12
4 400 185 650 185 650 255 400 255
5 497 10 598 19 604 102 549 165 447 141
7 320 165 380 165 380 285 510 285 510 305 420 305 320 305
4 300 55 360 45 380 135 320 145
4 200 35 280 35 280 215 200 215
5 170 215 300 265 300 325 180 305 150 245
6 90 35 130 95 170 55 180 155 120 195 70 135
4 400 125 430 145 410 175 380 155
4 90 225 140 225 130 265 90 285
5 540 280 541 280 800 280 800 400 540 400
10 380 340 381 340 520 340 460 380 540 440 720 460 420 460 140 440 140 300 280 380
4 80 300 81 300 140 280 120 360
621 162
```

Ausgabe

```
Dijkstra brauchte 5 Millisekunden.
Lisa muss anfangen zu rennen um 07:27:55
Lisa trifft den Bus um 07:30:41
Sie trifft den Bus an der y-Koordinate 340.055
Lisa braucht 00:02:46
```

```
Ihr Weg hat eine Laenge von 691.194
Sie geht ueber folgende Knoten:
Knoten (621.0|162.0) ID: L
Knoten (410.0|175.0) ID: P8
Knoten (380.0|165.0) ID: P3
Knoten (320.0|165.0) ID: P3
Knoten (280.0|215.0) ID: P5
Knoten (200.0|215.0) ID: P5
Knoten (170.0|215.0) ID: P6
Knoten (130.0|265.0) ID: P9
Knoten (0.0|340.055) ID: ENDE
```

Visuelle Darstellung



3.6 Eigenes Beispiel 1: Problem des Ray Casting Algorithmus

Falls der Mittelpunkt einer überprüften Strecke zwei Strecken gleichzeitig schneidet, so produziert der Ray Casting Algorithmus einen Fehler, weil dieser Schnitt doppelt gezählt ist. Während der Entwicklung habe ich folgendes Beispiel genutzt, um zu überprüfen, ob dieser richtig funktioniert. Wenn dies nicht der Fall ist, bildet sich im folgenden Beispiel ein rotes Kreuz innerhalb des Polygons. Dies ist bei unserem Algorithmus nicht der Fall:

Eingabe

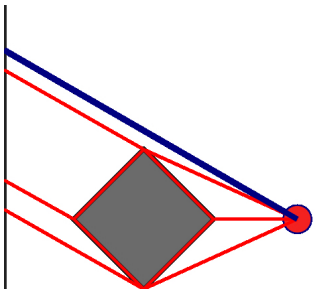
```
1
4 50 50 100 100 150 50 100 0
210 50
```

Ausgabe

```
Dijkstra brauchte 0 Millisekunden.
Lisa muss anfangen zu rennen um 07:29:22
Lisa trifft den Bus um 07:30:21
Sie trifft den Bus an der y-Koordinate 171.243
Lisa braucht 00:00:59
Ihr Weg hat eine Laenge von 242.486
Sie geht ueber folgende Knoten:
Knoten (210.0|50.0) ID: L
```

```
Knoten (0.0|171.243) ID: ENDE
```

Visuelle Darstellung (modifiziert, sodass alle Kanten des Graphen ausgegeben werden)



3.7 Eigenes Beispiel 2: Hindernisse mit gleichen Punkten

In der Aufgabenstellung wird nicht erläutert, welchen Körperbau Lisa hat. Doch wie könnte das relevant sein? Nun, wenn Lisa zwischen zwei Hindernissen durchgehen muss, die sehr nah aneinander liegen, wie z.B. zwei Polygone, die sich eine Seite teilen, oder zwei Polygone mit gleichem Punkt, so muss man wissen, ob Lisa zwischen ihnen durchkommt. Man könnte die Aufgabenstellung weiterführen, indem man überprüft, ob eine hinzuzufügende Kante einen bestimmten Mindestabstand zu allen anderen Kanten hat. Desweiteren könnte man einen gerichteten Graphen für die Modellierung der Aufgabenstellung nutzen, wenn Lisa z.B. in eine Richtung zu einem Knoten gehen kann, aber von dort aus nicht alle sichtbaren Punkte erreichen kann, sondern nur eine Teilmenge von ihnen, wie z.B. bei zwei Polygonen, die eine Ecke teilen. In der Aufgabenstellung wird angenommen, Lisa sei ein Punkt im Koordinatensystem - und deshalb unendlich dünn. Vielleicht ist sie ja vom ganzen Rennen schon abgemagert. Arme Lisa... Ich habe ihr aber schon eine gute Ernährungsberaterin empfohlen.

Das folgende Beispiel zeigt die Problematik auf.

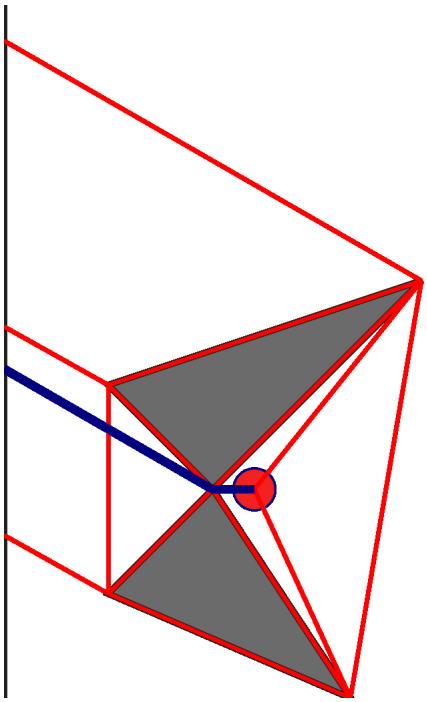
Eingabe

```
2
3 50 50 100 100 166 0
3 50 150 200 200 100 100
120 100
```

Ausgabe

```
Dijkstra brauchte 0 Millisekunden.
Lisa muss anfangen zu rennen um 07:29:46
Lisa trifft den Bus um 07:30:19
Sie trifft den Bus an der y-Koordinate 157.735
Lisa braucht 00:00:33
Ihr Weg hat eine Laenge von 135.47
Sie geht ueber folgende Knoten:
Knoten (120.0|100.0) ID: L
Knoten (100.0|100.0) ID: P1
Knoten (0.0|157.735) ID: ENDE
```

Visuelle Darstellung (modifiziert, sodass alle Kanten des Graphen ausgegeben werden)



3.8 Eigenes Beispiel 3: Knoten von Polygonen, die auf der y-Achse liegen

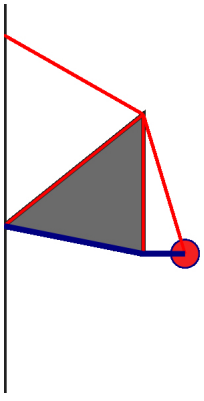
Wie in Abschnitt 1 erwähnt, kann es vorkommen, dass die kürzeste Route auf einem Polygon endet. Unser Programm berücksichtigt das.

```
1
3 0 120 100 200 100 100
130 100
```

Ausgabe

```
Dijkstra brauchte 0 Millisekunden.
Lisa muss anfangen zu rennen um 07:29:43
Lisa trifft den Bus um 07:30:14
Sie trifft den Bus an der y-Koordinate 120.0
Lisa braucht 00:00:31
Ihr Weg hat eine Laenge von 131.98
Sie geht ueber folgende Knoten:
Knoten (130.0|100.0) ID: L
Knoten (100.0|100.0) ID: P1
Knoten (0.0|120.0) ID: P1
```

Visuelle Darstellung (modifiziert, sodass alle Kanten des Graphen ausgegeben werden)



4 Quellcode

4.1 Die Klasse Main.java

```
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;

public class Main {
    static Point startingPoint;
    static Obstacle[] obstacles;
    static double angle;

    public static void main(String[] args) {
        //Einlesen der Kommandozeilenargumente
        if(args.length < 2){System.out.println("Benutzung: lisarennt EingabedateiPfad.txt
    AusgabedateiPfad [lisaGeschwindigkeit] [busGeschwindigkeit]"); System.exit(0);}
        String inputPath = args[0];
        String outputPath = args[1];
        double geschwindigkeitMaedchen = 15.0;
        double geschwindigkeitBus = 30.0;
        if(args.length == 4){
            geschwindigkeitMaedchen = Double.parseDouble(args[2]);
            geschwindigkeitBus = Double.parseDouble(args[3]);
        }
        //Errechnen des optimalen Winkels
        angle = computeAngle(geschwindigkeitMaedchen, geschwindigkeitBus);
        // Erstelle Graph
        WeightedUndirectedGraph graph = new WeightedUndirectedGraph(10);
        //Uebergebe Graph die Geschwindigkeiten
        graph.geschwindigkeitMaedchen = geschwindigkeitMaedchen;
        graph.geschwindigkeitBus = geschwindigkeitBus;
        int startingPointIndex = 0;
        //Beginne Daten einzulesen
        try {
            BufferedReader b = new BufferedReader(new FileReader(new File(inputPath)));
            // Lese Anzahl von Hindernissen
            int amountOfObstacles = Integer.parseInt(b.readLine());
            obstacles = new Obstacle[amountOfObstacles];
            for (int i = 0; i < amountOfObstacles; i++) {
                //Erstelle String Array mit Hinderniskoordinaten
                String[] l = b.readLine().split(" ");
                // Lese Anzahl von Punkten im Hindernis
                int amountOfPointsInObstacle = Integer.parseInt(l[0]);
                //Erstelle Hindernis mit noetiger Anzahl von Punkten
                obstacles[i] = new Obstacle(amountOfPointsInObstacle);
                Point firstPoint = new Point(Static.round(Double.parseDouble(l[1])),
    Static.round(Double.parseDouble(l[2])));
                //fuege ersten Knoten hinzu
                graph.addNode(firstPoint, i);
                //Der erste Punkt des Hindernisses ist der aktuelle Punkt, der nachste
    Punkt wird in der Schleife initialisiert und nach jeder Iteration erneuert
                Point currentPoint = firstPoint, nextPoint;
                for (int j = 3; j <= amountOfPointsInObstacle * 2; j += 2) {
                    // Erstellen des naechsten Knotens
                    nextPoint = new Point(Double.parseDouble(l[j]), Double.parseDouble(l[
    j + 1]));
                    //Fuege Knoten dem Graphen hinzu
                    graph.addNode(nextPoint, i);
                    //Verbinde aktuellen Knoten mit dem naechsten Knoten
                    obstacles[i].addLineSegment(new LineSegment(currentPoint, nextPoint))
    ;
                    currentPoint = nextPoint;
                }
                //Verbinde letzten Knoten des Hindernisses mit dem ersten Knoten
                obstacles[i].addLineSegment(new LineSegment(currentPoint, firstPoint));
            }
            String[] startingPointValues = b.readLine().split(" ");
            startingPoint = new Point(Double.parseDouble(startingPointValues[0]), Double.
    parseDouble(startingPointValues[1]));
            b.close();
        } catch (FileNotFoundException e) {
            System.out.println("Input-Datei konnte nicht gefunden werden.");
```

```
64              e.printStackTrace();
                System.exit(1);
66          } catch (IOException e) {
                System.out.println("Fehler beim Lesen der Daten.");
68              e.printStackTrace();
                System.exit(1);
70          } catch (Exception e) {
                System.out.println("Ein Fehler ist beim Lesen der Daten aufgetreten.");
72              e.printStackTrace();
                System.exit(1);
74          }
            // Beende Daten einzulesen
76
            //Beginne Sichtbarkeitsgraph-Algorithmus
78          //Beginne Sichtbarkeitsgraph-Algorithmus fuer P vereinigt mit S
            //Anzahl der Knoten im Graphen
80          int graphSize = 0;
            //Knoten (zum verbinden miteinander)
82          WeightedUndirectedGraph.Node[] nodes = null;
            try {
84              // Der Startpunkt wird dem Graphen (der Menge V) hinzugefuegt. Er gehoert zu
    keinem Hindernis
                startingPointIndex = graph.addNode(startingPoint, -1);
86              nodes = graph.vertices;
                graphSize = graph.size();
88              // Iteration durch alle Knoten des Graphen i
                for (int i = 0; i < graphSize; i++) {
90                  //Iteration durch alle Partnerknoten j
                    for (int j = 0; j < graphSize; j++) {
92                      // Knoten nicht mit sich selbst verbinden.
                        if (i == j) continue;
94                      //Gleiche (ueberlappende) Knoten nicht verbinden
                        if(nodes[i].point.equals(nodes[j].point)) continue;
96                      //Erstellen der Strecke IJ
                        LineSegment currentLineSegment = new LineSegment(nodes[i].point,
    nodes[j].point);
98                      //Per default schneidet die Strecke kein Polygon.
                        //falls Punkt I und Punkt J im gleichen Hindernis sind, wird
    inSameObstacle true
100                     boolean intersects = false, inSameObstacle = (nodes[i].obstacleIndex
    == nodes[j].obstacleIndex);
                        //Ueberpruefung, ob IJ im Hindernis liegt, falls I und J teil des
    gleichen Hindernisses sind. Falls ja, verwerfe IJ
102                     if (inSameObstacle && obstacles[nodes[i].obstacleIndex].
    isInsideObstacle(currentLineSegment))
                            continue;
104                     intersects = false;
                        for (int k = 0; k < obstacles.length; k++) {
106                         //Falls IJ ein Hindernis schneidet, intersects auf true und
    Schleife beenden.
                            if (obstacles[k].intersects(currentLineSegment)) {
108                             intersects = true;
                                break;
110                         }
                        }
112                     //Nur wenn IJ keines der Hindernisse schneidet, werden I und J im
    Graphen verbunden
                        if (!intersects) {
114                         graph.connect(i, j, nodes[i].point.distance(nodes[j].point));
                        }
116                 }
                }
118         } catch (Exception e) {
                e.printStackTrace();
120         }
            //Beende Sichtbarkeitsgraph-Algorithmus fuer P vereinigt mit S
122         //Beginne Sichtbarkeitsgraph-Algorithmus fuer X

124         //Wir iterieren durch alle Knoten und versuchen, Knoten I unter dem
    effizientesten Winkel mit der y-Achse zu verbinden
            for (int i = 0; i < graphSize; i++) {
126             //errechnen der y-Koordinate anhand von I's x-Koordinate und dem
    effizientesten Winkel
```

```
            double yCoordinate = Static.round(nodes[i].point.y + computeYShift(nodes[i].
    point.x));
128             //Erstelle y-Schnittpunkt
                Point newPoint = new Point(0, yCoordinate);
130             //erstelle Strecke von I zum y-Schnittpunkt
                LineSegment currentSegmentToStreet = new LineSegment(nodes[i].point, newPoint
    );
132             //ueberpruefe, ob Strecke von I zum moeglichen y-Schnittpunkt kein Hindernis
    schneidet
                boolean intersects = false;
134             //Iteration durch alle Hindernisse
                for (int j = 0; j < obstacles.length; j++) {
136                 //Iteration durch alle Strecken der Hindernisse
                    if (obstacles[j].intersects(currentSegmentToStreet)) {
138                     intersects = true;
                        break;
140                 }
                }
142             //Falls die Strecke kein Hindernis schneidet, fuege sie dem Graphen hinzu
                if (!intersects) {
144                 try {
                        int a = graph.addNode(newPoint, -1);
146                     graph.connect(i, a, Static.round(nodes[i].point.distance(newPoint)));
                    } catch (Exception e) {
148                     e.printStackTrace();
                    }
150             }

152         }
            //Beende Sichtbarkeitsgraph-Algorithmus fuer X
154         //Beende Sichtbarkeitsgraph-Algorithmus

156     nodes = graph.vertices;
            //Beginne Algorithmus von Dijkstra
158         graph.dijkstra(startingPointIndex);
            //Beende Algorithmus von Dijkstra
160         //Beginne Suche des optimalen Knotens
            ArrayList<Integer> shortestPath = graph.getShortestPath();
162         //Beende Suche des optimalen Knotens
            //Beginne Ausgabe der Ergebnisse
164         DateTimeFormatter format = DateTimeFormatter.ISO_LOCAL_TIME;
            LocalTime t = LocalTime.parse("07:30:00");
166         LocalTime leavingTime, meetingTime;
            double latestLeavingTime = Math.round(graph.getLatestLeavingTime());
168         double meetingTimeNum = Math.round(graph.getMeetingTime());
            leavingTime=t.plusSeconds((long) latestLeavingTime);
170         meetingTime = t.plusSeconds((long) meetingTimeNum);
            System.out.println("Lisa␣muss␣anfangen␣zu␣rennen␣um␣"+ format.format(leavingTime)
    );
172         System.out.println("Lisa␣trifft␣den␣Bus␣um␣" + format.format(meetingTime));
            System.out.println("Sie␣trifft␣den␣Bus␣an␣der␣y-Koordinate␣" + graph.
    getMeetingYCoord());
174         System.out.println("Lisa␣braucht␣" + format.format(meetingTime.minusSeconds(
    leavingTime.toSecondOfDay())));
            System.out.println("Ihr␣Weg␣hat␣eine␣Laenge␣von␣" + Static.round(nodes[graph.
    getShortestPathNodeIndex()].distanceToRoot));
176         System.out.println("Sie␣geht␣ueber␣folgende␣Knoten:");
            int counter = 0;
178         for(Integer index: shortestPath){
                System.out.println("Knoten␣" + nodes[index].point + "␣ID:" + (nodes[index].
    obstacleIndex==-1?counter==0?"␣L":counter==shortestPath.size()-1?"␣ENDE":"␣P"+(nodes[
    index].obstacleIndex+1):"␣P"+(nodes[index].obstacleIndex+1)));
180             counter++;
            }
182         SVGOutputter s = new SVGOutputter(graph, obstacles, startingPoint, shortestPath);
            s.constructDocument();
184         s.outputSVG(outputPath+".svg");
    }
186
    public static double computeAngle(double geschwindigkeitMaedchen, double
    geschwindigkeitBus) {
188     //unsere Formel
```

```
        return Static.round(Math.toDegrees(Math.acos(Math.sqrt(-1 * Math.pow((
    geschwindigkeitBus / geschwindigkeitMaedchen), -2) + 1))));
190     }

192     public static double computeYShift(double xDistance) {
            //noetiger y-Shift mit Tangens
194         return Static.round(Math.abs(Math.tan(Math.toRadians(angle)) * xDistance));
        }
196 }
```

i

4.2 Die Klasse WeightedUndirectedGraph.java

```
import java.util.ArrayList;
2
public class WeightedUndirectedGraph{
4     private int CURRENT_CAPACITY;
       Node[] vertices;
6      Edge[][] adjMat;
      private int size;
8     double geschwindigkeitMaedchen;
      double geschwindigkeitBus;
10    private double latestLeavingTime;
      private double meetingTime;
12    private double meetingYCoord;
      private int shortestPathNodeIndex;
14    public WeightedUndirectedGraph(int size){
          this.CURRENT_CAPACITY = size;
16        this.size = -1;
          this.adjMat = new Edge[size][size];
18        this.vertices = new Node[size];
          this.latestLeavingTime =Double.NEGATIVE_INFINITY;
20    }

22    public void connect(int startNode, int endNode, double weight) throws Exception{
          if(vertices[startNode] != null && vertices[endNode]!= null) {
24            Edge e = new Edge(weight);
              adjMat[startNode][endNode] = e;
26            adjMat[endNode][startNode] = e;
          }else{
28            throw new Exception("Cannot␣connect:␣Node␣doesn't␣exist.");
          }
30    }
      public void disconnect(int startNode, int endNode) throws Exception{
32        if(vertices[startNode] != null && vertices[endNode]!= null) {
              adjMat[startNode][endNode] = null;
34            adjMat[endNode][startNode] = null;
          }else{
36            throw new Exception("Cannot␣disconnect:␣Node␣doesn't␣exist.");
          }
38    }
      public boolean isConnected(int startNode, int endNode){
40        return vertices[startNode] != null && vertices[endNode]!= null || startNode ==
    endNode;
      }
42    public double getWeight(int startNode, int endNode)throws Exception{
          if(vertices[startNode] != null && vertices[endNode]!= null && adjMat[startNode][
    endNode]!= null) {
44            return adjMat[startNode][endNode].getWeight();
          }
46        else{
              throw new Exception("Cannot␣get␣weight:␣Node␣or␣edge␣doesn't␣exist.␣startNode
    :␣" + startNode+ ",␣endNode:␣"+ endNode);
48        }
      }
50    public int addNode(Point a, int oIndex) throws Exception {
          if (size+1 <= CURRENT_CAPACITY/2) {
52            vertices[++size] = new Node(a, oIndex);
              return size;
54        } else {
              Node[] currentVertices = this.vertices;
```

```
            CURRENT_CAPACITY*=2;
            vertices = new Node[CURRENT_CAPACITY];
            for(int i = 0; i < currentVertices.length; i++){
                if(currentVertices[i] != null) vertices[i] = currentVertices[i];
                else break;
            }
            Edge[][] currentAdjMat = adjMat;
            adjMat = new Edge[CURRENT_CAPACITY][CURRENT_CAPACITY];
            for(int i = 0; i < currentAdjMat.length; i++){
                for(int j = 0; j< currentAdjMat.length; j++){
                    adjMat[i][j] = currentAdjMat[i][j];
                }
            }
            vertices[++size] = new Node(a, oIndex);
            return size;
        }
    }
    public int findUnvisitedNeighbor(int index){
        for(int i = vertices[index].lastUncheckedNode; i <= size; i++){
            if(adjMat[index][i]!= null){
                vertices[index].lastUncheckedNode = i+1;
                return i;
            }
        }
        return -1;
    }
    public void dijkstra(int startIndex){
        long startTime = System.currentTimeMillis();
        Set sptSet = new Set(size+1);
        sptSet.add(startIndex);
        int v = startIndex;
        vertices[startIndex].distanceToRoot = 0;
        vertices[startIndex].path.add(startIndex);
        int u = -1;
        int current;
        double currentDistance;
        while(sptSet.size() < this.size()) {
            while ((current = this.findUnvisitedNeighbor(v)) != -1) {
                if(vertices[current].distanceToRoot > (currentDistance = vertices[v].
distanceToRoot + adjMat[v][current].getWeight()) || vertices[current].distanceToRoot
== -1){
                    vertices[current].distanceToRoot = currentDistance;
                    vertices[current].path = (ArrayList) vertices[v].path.clone();
                    vertices[current].path.add(current);
                }
            }
            currentDistance = -1;
            for(int i = 0; i <= this.size; i++){
                if(!sptSet.contains(i) && (currentDistance == -1 || (vertices[i].
distanceToRoot < currentDistance && vertices[i].distanceToRoot >= 0))) {
                    currentDistance = vertices[i].distanceToRoot;
                    u = i;
                }
            }
            sptSet.add(u);
            v = u;
        }
        System.out.println("Dijkstra␣brauchte␣" + (System.currentTimeMillis()-startTime)
+ "␣Millisekunden.");
    }

    public ArrayList<Integer> getShortestPath(){
        ArrayList<Integer> shortestPath = null;
        double currentLeavingTime = 0.0;
        for(int i = 0; i < this.size(); i++){
            if( vertices[i].point.x == 0 &&(currentLeavingTime = vertices[i].
determineLeavingTime()) > latestLeavingTime){ latestLeavingTime = currentLeavingTime;
            shortestPath = vertices[i].path;
            meetingTime = vertices[i].point.y/(geschwindigkeitBus/3.6);
            meetingYCoord = vertices[i].point.y;
            shortestPathNodeIndex = i;
            }
        }
```

```
124         return shortestPath;
    }
126     public double getLatestLeavingTime(){
        return this.latestLeavingTime;
128     }
    public double getMeetingTime(){
130         return this.meetingTime;
    }
132     public double getMeetingYCoord(){
        return this.meetingYCoord;
134     }
    public int getShortestPathNodeIndex(){
136         return this.shortestPathNodeIndex;
    }
138 
    public void resetNodes(){
140         for(int i = 0; i < vertices.length; i++){
            vertices[i].distanceToRoot = -1;
142             vertices[i].lastUncheckedNode = 0;
            vertices[i].path = new ArrayList<Integer>();
144         }
    }
146 
    class Node{
148         Point point;
        int lastUncheckedNode;
150         double distanceToRoot;
        int obstacleIndex;
152         ArrayList<Integer> path;
        public Node(Point p, int oIndex){
154             obstacleIndex = oIndex;
            this.point = p;
156             this.lastUncheckedNode = 0;
            path = new ArrayList<Integer>();
158             distanceToRoot = -1;
        }
160         @Override
        public String toString(){
162             return "Node:␣" + point + "␣Distance␣to␣root␣"+ this.distanceToRoot + "␣have␣
    to␣start␣walking␣after␣" + determineLeavingTime() + "␣secs.";
        }
164         public double determineLeavingTime(){
            return this.point.y/(geschwindigkeitBus/3.6)-this.distanceToRoot/(
    geschwindigkeitMaedchen/3.6);
166         }
    }
168      class Edge{
        private double WEIGHT;
170         private boolean visited;
        public Edge(double weight){
172             this.WEIGHT = weight;
            this.visited = false;
174         }
        public double getWeight(){
176             return WEIGHT;
        }
178         public void setVisited(boolean visited){
            this.visited = visited;
180         }
        public boolean getVisited(){
182             return visited;
        }
184     }
    public int size(){return size+1;}
186 
}
```

4.3 Die Klasse LineSegment.java

```java
public class LineSegment {
    Point start, end;
    double digits = 4;
    static double errorTolerance;
    public LineSegment(Point start, Point end){
        if(start == null) throw new IllegalArgumentException("Starting point was null.");
        if(end == null) throw new IllegalArgumentException("Ending point was null.");
        this.start = start;
        this.end = end;
        errorTolerance = Math.pow(10.0, digits);

    }
    public boolean intersects(LineSegment ls){

        if(this.start.matches(ls.start)||this.start.matches(ls.end)||this.end.matches(ls.
    start)||this.end.matches(ls.end)) return false;
        int a = orientation(this.start, this.end, ls.start);
        int b = orientation(this.start, this.end, ls.end);
        int c = orientation(ls.start, ls.end, this.start);
        int d = orientation(ls.start, ls.end, this.end);
        if(a != b && c != d) {
            return true;
        }
        return false;
    }

    private int orientation(Point a, Point b, Point c){
        double val = Static.round((b.y-a.y)*(c.x-b.x)-(b.x-a.x)*(c.y-b.y));
        if (val == 0)
            return 0;
        else if(val < 0)
            return 2;
        return 1;
    }
    public boolean matches(LineSegment a){
        return (this.start.matches(a.start) && this.end.matches(a.end)) || (this.end.
    matches(a.start)&& this.start.matches(a.end));
    }

    @Override
    public String toString(){
    return start + " --> " + end;
    }

}
```

4.4 Die Klasse Obstacle.java

```java

public class Obstacle {
    LineSegment[] lineSegments;
    int amountOfPoints;
    int currentCapacity;
    public Obstacle(int amountOfPoints){
        this.amountOfPoints = amountOfPoints;
        lineSegments = new LineSegment[amountOfPoints];
        currentCapacity = 0;
    }
    public void addLineSegment(LineSegment ls) throws Exception {
        if(currentCapacity == lineSegments.length) throw new Exception("Obstacle exceeded
    original capacity at LineSegment" + ls);
        lineSegments[currentCapacity++] = ls;
    }
    public boolean intersects(LineSegment l){
        for(int i = 0; i < lineSegments.length; i++){
            if(l.intersects(lineSegments[i])) {
                return true;
            }
        }
```

```
        return false;
    }
    public boolean isInsideObstacle(LineSegment l){
        double xCoordinateMiddle = Static.round((l.start.x+l.end.x)/2), yCoordinateMiddle
     = Static.round((l.start.y+l.end.y)/2);
        Point middlePoint = new Point(xCoordinateMiddle, yCoordinateMiddle);
        LineSegment ray = new LineSegment(middlePoint, new Point(10e3, middlePoint.y));
        int intersections = countIntersections(ray, l);
        return intersections==-1?false:intersections%2!=0?true:false;
    }
    public int countIntersections(LineSegment ray, LineSegment l){
        int amountOfIntersections = 0;
        boolean currentTouches = false, previousTouches = false;
        for(int i = 0; i < lineSegments.length; i++){
            currentTouches = rayTouchesLineSegment(ray, lineSegments[i]);
            if(l.matches(lineSegments[i])) return -1;
            if(lineSegments[i].intersects(ray) && !(currentTouches && previousTouches)) {
    amountOfIntersections++;}
            previousTouches = currentTouches;
        }
        return amountOfIntersections;
    }

    private boolean rayTouchesLineSegment(LineSegment ray, LineSegment ls){
        return (ls.intersects(ray) && (ls.start.y == ray.start.y || ls.end.y == ray.start
    .y));
    }
}
```

4.5 Die Klasse Point.java

```
public class Point {
    double x, y;
    public Point(double x, double y){
        this.x =  x;
        this.y =  y;
    }
    public double distance(Point p){
        return Static.round(Math.hypot(this.x-p.x, this.y - p.y));
    }
    @Override
    public String toString(){
        return "("+this.x+"|"+this.y+")";
    }
    public boolean matches(Point a){
        return Static.round(this.x) == Static.round(a.x) && Static.round(this.y) ==
    Static.round(a.y);
    }
}
```

4.6 Die Klasse Set.java

```
//Speicherkomplexitaet: n.
public class Set {
    private int size;
    private boolean[] items;
    public Set(int initialSize){
        items = new boolean[initialSize];
        size = 0;
    }
    public void add(int item){
        if(item > items.length || item < 0) throw new IllegalArgumentException("Cannot␣
    add␣item␣to␣set:␣invalid␣index");
        items[item] = true;
```

```
        size++;
    }
    public boolean contains(int item){
        return items[item];
    }
    public int size(){
        return size;
    }
}
```

4.7 Die Klasse Static.java

```
public class Static {
    static double digits = 3.0;
    static double tolerance = Math.pow(10.0, digits);
    public static double round(double d){
        return Math.floor(d* tolerance)/ tolerance;
    }
}
```

5 Benutzung

Für die Ausführung muss eine aktuelle Java-Version (am besten 8 oder neuer) auf dem Gerät installiert sein.
Das Programm lisarennt.jar kann über die Kommandozeile folgendermaßen ausgeführt werden:

```
java -jar lisarennt.jar DateipfadDerEingabedatei.txt SVGDateiname
```

Optional können andere Geschwindigkeiten als 15 km/h für Lisa und 30 km/h für den Bus gesetzt werden. In diesem Falle:

```
java -jar lisarennt.jar DateipfadDerEingabedatei.txt SVGDateinameUndPfad
    GeschwindigkeitLisa GeschwindigkeitBus
```