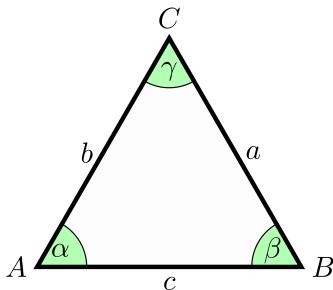


Dokumentation zur Aufgabe „Lisa rennt“

Die Lösungsidee und Umsetzung:

Der schnellste Weg durch die Hindernisse führt zwangsweise an den Ecken der Hindernisse entlang. Dies lässt sich einfach am Beispiel eines Dreiecks erklären:



Der schnellste Weg von Punkt A nach Punkt C führt über die Gerade b statt beispielsweise über die Geraden c und a. Folglich ist es nicht sinnvoll sich erst zu einer Kante eines Polygons zu bewegen um sich von dort aus weiter zu einer Ecke zu bewegen, da dies zwangsweise nicht der schnellste Weg ist.

Daraus folgt, dass zur Lösung der Aufgabe eine Art Weg-Netz zwischen allen Ecken der Polygone erschaffen werden kann, welches dann zwangsweise den schnellsten Weg beinhaltet.

Das Weg-Netz soll in der Implementation wie folgt dargestellt sein: für jede Ecke wird ein Array mit der Größe „Anzahl aller Polygon-Ecken“ erstellt, welcher an jeder Position die Information beinhaltet, ob eine Verbindung zu dem Knoten mit dem entsprechenden Index besteht und sollte dies der Fall sein zudem wie weit die beiden Knoten von einander entfernt liegen. Diese Arrays wiederum sollen alle zusammen in einem Array gemeinsam abgespeichert werden, sodass ein 2-dimensionaler Array entsteht der an der Position [a][b] das selbe beinhaltet wie an der Stelle [b][a]. Ist das Weg-Netz erst einmal gelegt fehlt nur noch eine Information: die möglichen Ziel-Ecken bzw. Ziel-Knoten.

Diese sind all jene Knoten, welche eine direkte Verbindung zur y-Achse über die Gerade mit optimaler Steigung zum Laufen haben.

Nun die Frage: „Was ist die optimale Steigung, mit der Lisa auf den Bus zulaufen kann?“ Lisa könnte frontal Richtung Bus laufen, aber dies würde ihr nicht das beste Ergebnis bringen. Jede Bewegung nach unten ist zudem schlecht, da der Bus schneller ist als sie und ihre Startzeit sich um so weiter nach vorne verschieben würde, je weiter sie unten auf der y-Achse ankommt, da der Bus bis zu diesem Punkt zwangsweise weniger Zeit braucht als bis zu einem Punkt der weiter oben auf der y-Achse liegt. Unter zwei Wegen mit gleicher Länge ist zwangsweise der Weg besser für Lisa, der weiter oben auf der y-Achse ankommt.

Also zurück zur Frage was die optimale Steigung ist, mit der Lisa zum Bus laufen kann: ich kann es mathematisch nicht erklären, doch durch ausknobeln bin ich zu der Lösung gekommen, dass das beste Ergebnis erzielt werden kann, wenn Lisa mit einer Steigung von $-\sqrt{3}/3$ läuft. Aufgrund fehlender mathematischer Beweisbarkeit findet sich auf der folgenden Seite eine Tabelle mit den Ergebnissen meiner ausknoblen Arbeit.

Stehen nun alle Informationen fest muss nur noch der schnellste Weg gefunden werden. Dies geschieht, indem der Dijkstra-Algorithmus verwendet wird mit Lisas Haus als Startknoten. Nachdem der Algorithmus die schnellsten Wege für jeden Knoten bestimmt hat kann anschließend für alle zuvor ausgesuchten Ziel-Knoten die exakte Zeit berechnet werden, wann Lisa losfahren müsste und das beste Ergebnis kann ausgegeben werden.

Die optimale Steigung mit der sich Lisa bewegen kann, habe ich herausgefunden, indem ich die verschiedenen Startzeiten für verschiedene Steigungen ausgerechnet habe. Dabei habe ich den Bus auf (0|0) starten lassen und Lisa auf (1|2). Lisa und der Bus treffen sich, wenn Lisa gerade auf den Bus zufährt, aufgrund dessen, dass der Bus genau doppelt so schnell ist wie Lisa, bei dem Punkt (0|2), wenn Bus und Lisa zur selben Zeit starten.

Lisas Geschwindigkeit: $15\text{km/h} = 4,16667\text{m/s}$
 Geschwindigkeit vom Bus: $30\text{km/h} = 8,33333\text{m/s}$

An dieser Stelle etwas Physik: $s = v * t \rightarrow$ also: $t = s / v$

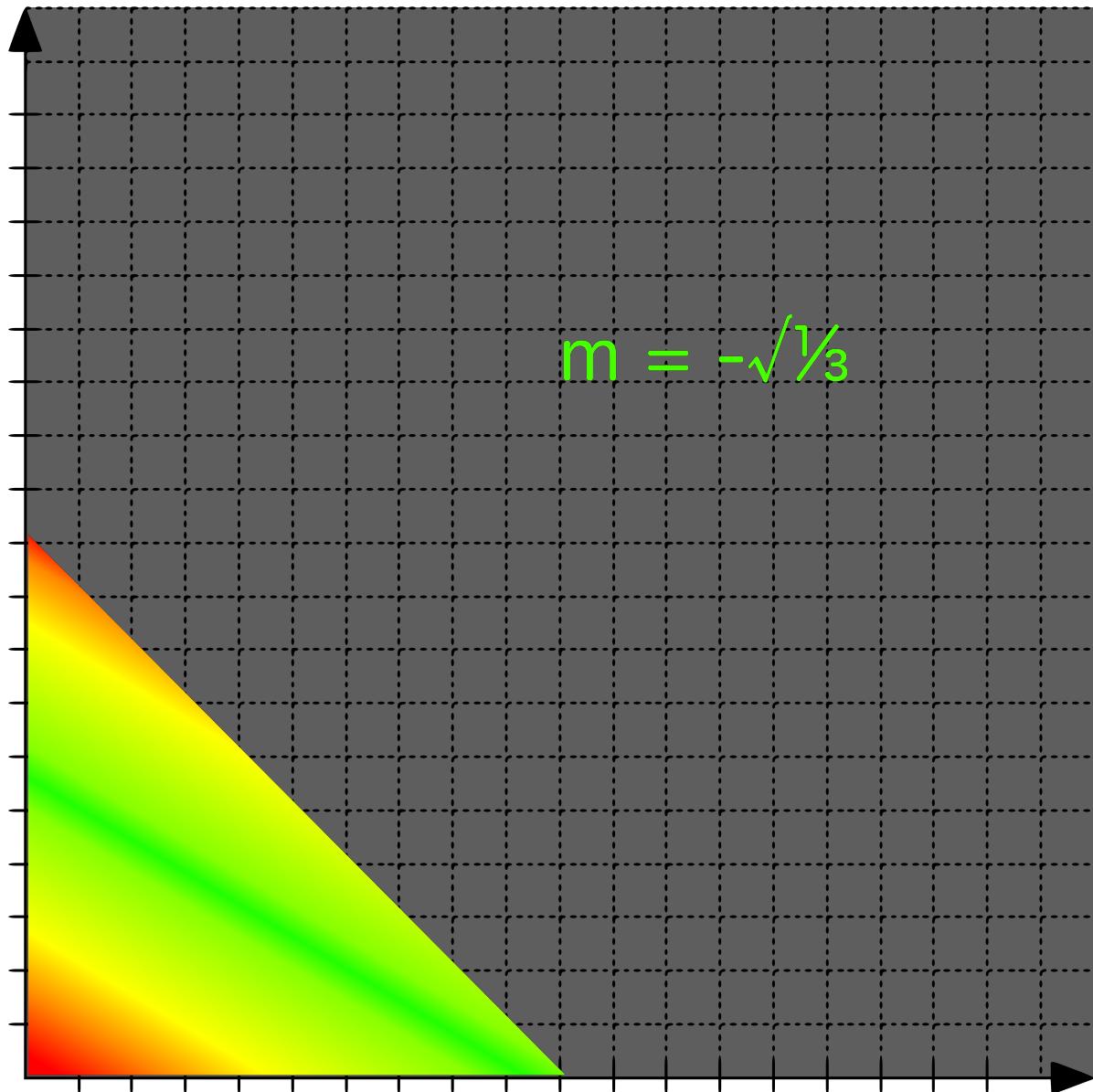
Somit lässt sich die Zeit die Lisa benötigt berechnen durch das Teilen der zweiten Spalte durch $4,16667\text{m/s}$ und die Zeit die der Bus benötigt durch das Teilen der 3. Spalte durch $8,33333\text{m/s}$.

y des Treffpunktes	Lisas Strecke (in m)	Strecke des Busses (in m)	Benötigte Zeit von Lisa (in s)	Benötigte Zeit vom Bus (in s)	Zeit zwischen Lisas Start und dem Start des Busses (in s)
2	1	2	0,24	0,24	0
2,4	$\sqrt{1+0,4^2}$	2,4	0,25849	0,288	0,02951
2,5	$\sqrt{1+0,5^2}$	2,5	0,26833	0,3	0,03167
2,55	$\sqrt{1+0,55^2}$	2,55	0,27391	0,306	0,03210
2,56	$\sqrt{1+0,56^2}$	2,56	0,27507	0,3072	0,03213
2,57	$\sqrt{1+0,57^2}$	2,57	0,27625	0,3084	0,03215
2,576	$\sqrt{1+0,576^2}$	2,576	0,27697	0,30912	0,032156100262
2,577	$\sqrt{1+0,577^2}$	2,577	0,27709	0,30924	0,032156233895
$2+\sqrt{1/3}$	$\sqrt{1+1/3}$	$2+\sqrt{1/3}$	0,27713	0,30928	0,032156243812
2,578	$\sqrt{1+0,578^2}$	2,578	0,27721	0,30936	0,032156211573
2,58	$\sqrt{1+0,58^2}$	2,58	0,27745	0,3096	0,03216
2,59	$\sqrt{1+0,59^2}$	2,59	0,27866	0,3108	0,03214
2,6	$\sqrt{1+0,6^2}$	2,6	0,27989	0,312	0,03212
2,7	$\sqrt{1+0,7^2}$	2,7	0,29296	0,324	0,03104
3	$\sqrt{2}$	3	0,33941	0,36	0,02059
4	$\sqrt{5}$	4	0,53666	0,48	-0,05666

Die letzte Spalte ergibt sich, indem von der benötigten Zeit vom Bus die benötigte Zeit von Lisa abgezogen wird. So bleibt die Zeit über, die Lisa warten kann, während der Bus bereits losgefahren ist.

Als ich die Zahl 0,577 ausgeknobelt hatte konnte ich immer noch nicht erklären, wie diese Zahl zu Stande kam als optimale Steigung für Lisa. Also habe ich geguckt wie die Zahl entstehen kann und habe beim Quadrieren festgestellt, dass die Zahl 0,577 sehr dicht an der Quadratwurzel von 1/3 liegt. Diese habe ich dann testweise eingesetzt als Steigung und bin zu dem Schluss gekommen, dass dies tatsächlich die optimale Steigung für Lisas Weg zu sein scheint.

Das ganze ist hier noch einmal graphisch dargestellt:



Die untere rechte Ecke des Dreiecks stellt Lisas derzeitigen Standpunkt dar. Die grüne Linie zeigt den besten Weg den Lisa mit den angegebenen Geschwindigkeiten laufen kann um möglichst spät los zu müssen.

Quelltext und Umsetzung

Die Umsetzung der Lösungsidee erfolgt in meiner Bearbeitung in der Programmiersprache Ruby. Das Programm beziehungsweise Script („lisa_rennt.rb“) liegt im Ordner „Aufgabe 1\Programm“ und stellt zeitgleich Quelltext und ausführbare Datei dar. Ich habe das Script mit der Ruby Version 2.5.3-1-x64 in der CMD von Windows 10 getestet.

Die nachfolgenden Methoden sind teilweise nur dazu da die Lösungen einzelner Gleichungen zu erhalten. Aus diesem Grund findet sich bei diesen Methoden nur der Methoden Name statt die gesamte Methode.

Die Variablen Namen in dem Programm entsprechen denen, die bei uns im Mathematik Unterricht verwendet wurden:

x: Wert auf der x-Achse im Koordinatensystem
y: Wert auf der y-Achse im Koordinatensystem
m: Steigung einer Gerade
n: y-Achsenabschnitt einer Geraden

Es gilt also $y = m \cdot x + n$.

def gleichsetzenVonGeraden(m1, n1, m2, n2)

Diese Methode findet den x Punkt an dem zwei Geraden sich schneiden. Sollten die zwei übergebenen Geraden die selbe Steigung haben wird entweder ∞ oder nil zurückgegeben, je nachdem ob die Geraden den selben y-Achsenabschnitt haben oder nicht.

Nimmt man jeweils die rechten Hälften von $y = m_1 \cdot x + n_1$ und $y = m_2 \cdot x + n_2$ und setzt diese gleich, sodass die Formel $m_1 \cdot x + n_1 = m_2 \cdot x + n_2$ entsteht, kann man diese zu $x = (n_1 - n_2) / (m_2 - m_1)$ umformen. In diese werden letztlich die übergebenen Zahlen eingesetzt, wodurch der x Wert des Schnittpunktes der zwei Geraden bestimmt werden kann.

def xDurchYBestimmen(y, m, n)

Es wird der x Wert einer Geraden anhand der Werte von y, m und n berechnet. Dies geschieht durch die Formel $x = (y - n) / m$, welche aus $y = m \cdot x + n$ entstanden ist.

def yDurchXBestimmen(x, m, n)

Es wird der y Wert einer Geraden anhand der Werte von x, m und n berechnet. Dies geschieht durch einsetzen der übergebenen Werte in die Formel $y = m \cdot x + n$.

def steigungDurchZweiPunkteBestimmen(x1, y1, x2, y2)

Es werden zwei Punkte übergeben. Die Rückgabe ist die Steigung der Geraden, welche durch diese beiden Punkte verläuft. Sind die Werte x1 und x2 identisch ist die Steigung nicht bestimmbar und es wird nil zurückgegeben. Ansonsten ist die Rückgabe das Ergebnis des Einsetzens der Werte in die Formel $m = (y_2 - y_1) / (x_2 - x_1)$.

def yAchsenabschnittBestimmen(x, y, m)

Diese Methode gibt den zu den übergebenen Werten gehörenden y-Achsenabschnitt der Gleichung zurück. Dies geschieht durch die aus der Formel $y = m \cdot x + n$ entstandenen Formel $n = y - m \cdot x$.

Die folgende Methode kontrolliert, ob die Gerade, die durch die ersten beiden übergebenen Punkte verläuft (x_1, y_1, x_2, y_2) sich mit der Geraden schneidet, welche durch die anderen beiden übergebenen Punkte (x_3, y_3, x_4, y_4) geht. Und zwar innerhalb der Bereiche der jeweiligen Punkte.

```
def schneidetInBereich(x1, y1, x2, y2, x3, y3, x4, y4)
    m1 = steigungDurchZweiPunkteBestimmen(x1, y1, x2, y2)
    m2 = steigungDurchZweiPunkteBestimmen(x3, y3, x4, y4)
    if [x1, y1] == [x3, y3] or [x2, y2] == [x3, y3] or [x1, y1] == [x4, y4] or [x2, y2] == [x4, y4]
        return nil
    end
```

Als erstes werden die Steigungen der Geraden zwischen den ersten und letzten beiden Punkten berechnet. Sollten der erste und dritte, zweite und dritte, erste und vierte oder der zweite und vierte Punkt gleich sein wird zudem nil zurückgegeben. In diesem Fall müssten die Geraden zum selben Polygon gehören (davon ausgehend das wie in allen Beispielen Polygone einander nie berühren) und dies wird später gesondert betrachtet.

```
if m1 == nil
    if m2 == nil
        if x1 == x3 and ((y1..y2).cover? y3 or (y2..y1).cover? y3) or ((y3..y4).cover? y1 or
(y4..y3).cover? y1))
            return true
        else
            return false
    end
```

Falls sowohl m1=nil wie auch m2=nil gilt - also beide Geraden gerade nach oben gehen - wird kontrolliert, ob die Geraden auf der selben x Koordinate liegen und sich in ihren y-Bereichen schneiden. Ist dies der Fall wird true zurückgegeben, andernfalls false.

```
else
    n2 = yAchsenabschnittBestimmen(x3, y3, m2)
    y = yDurchXBestimmen(x1, m2, n2)
    if ((y1..y2).cover? y or (y2..y1).cover? y) and ((y3..y4).cover? y or (y4..y3).cover? y)
and ((x3..x4).cover? x1 or (x4..x3).cover? x1)
        return true
    else
        return false
    end
end
```

Sollte m1=nil sein, aber m2≠nil ist, wird kontrolliert, ob die Gerade, die zu m2 gehört, an der x Koordinate der anderen Gerade einen y Wert hat welcher innerhalb des Bereiches von y3 bis y4 liegt sowie der x Wert der ersten Gleichung innerhalb des Bereiches von x3 bis x4 liegt. Ist dies der Fall wird true zurückgegeben, andernfalls false.

```

else
  if m2 == nil
    n1 = yAchsenabschnittBestimmen(x1, y1, m1)
    y = yDurchXBestimmen(x3, m1, n1)
    if ((y1..y2).cover? y or (y2..y1).cover? y) and ((y3..y4).cover? y or (y4..y3).cover? y)
and ((x1..x2).cover? x3 or (x2..x1).cover? x3)
      return true
    else
      return false
  end

```

Sollte $m1 \neq \text{nil}$ sein, aber $m2 = \text{nil}$ ist, wird kontrolliert, ob die Gerade, die zu $m1$ gehört, an der x Koordinate der anderen Gerade einen y Wert hat welcher innerhalb des Bereiches von $y1$ bis $y2$ liegt sowie der x Wert der zweiten Gleichung innerhalb des Bereiches von $x1$ bis $x2$ liegt. Ist dies der Fall wird true zurückgegeben, andernfalls false.

```

else
  n1 = yAchsenabschnittBestimmen(x1, y1, m1)
  n2 = yAchsenabschnittBestimmen(x3, y3, m2)
  x = gleichsetzenVonGeraden(m1, n1, m2, n2)
  if x == nil
    return false
  elsif x == Float::INFINITY
    if (x1..x2).cover? x3 or (x2..x1).cover? x3 or (x1..x2).cover? x4 or
(x2..x1).cover? x4
      return true
    else
      return false
    end
  else
    y = yDurchXBestimmen(x, m1, n1)
    if ((x1..x2).cover? x or (x2..x1).cover? x) and ((x3..x4).cover? x or
(x4..x3).cover? x)
      return true
    else
      return false
    end
  end
end
end

```

Wenn sowohl $m1$ wie auch $m2 \neq \text{nil}$ sind, werden die beiden Geraden gleichgesetzt und es wird kontrolliert, ob der zurückgegebene x Wert im Bereich $x1$ bis $x2$ sowie im Bereich $x3$ bis $x4$ liegt. Ist dies der Fall wird true zurückgegeben, andernfalls false. Sollte beim Gleichsetzen nil zurückgegeben werden, wird ebenfalls false zurückgegeben. Sollte ∞ zurückgegeben werden wird kontrolliert, ob $x3$ oder $x4$ im Bereich von $x1$ bis $x2$ liegt. Ist dies der Fall wird true zurückgegeben, andernfalls false.

```
$svgPolygone = ""
```

```
def polygoneEinlesen(datei)
```

Diese Methode liest die Eckpunkte der Polygone sowie die Position von Lisas Haus ein und speichert diese in einem Array namens knoten. Jeder Knoten in diesem Array besteht aus der x und y Koordinate sowie aus der id des zugehörigen Polygons. Lisas Haus hat hier zunächst die id 0.

Des Weiteren werden alle Linien der Polygone in einem weiteren Array namens polygonLinien gespeichert, welche je aus zwei x Koordinaten, zwei y Koordinaten (x_1, y_1, x_2, y_2) und der id des zugehörigen Polygons bestehen.

Die Arrays knoten und polygonLinien werden von der Methode zurückgegeben.

Zugleich werden die Polygone als SVG Code in dem String \$svgPolygone gespeichert für die spätere visuelle Darstellung.

```
def sindPunkteNachbarn(polygonLinien, id, x1, y1, x2, y2)
```

Die Methode kontrolliert, ob zwei Punkte im Array polygonLinien als Linie wieder zu finden sind und folglich benachbart sind. Sind die übergebenen Punkte benachbart, wird true zurückgegeben, andernfalls false.

```
def punktLiegtAufLinie(x1, y1, x2, y2, x3, y3)
```

```
    m = steigungDurchZweiPunkteBestimmen(x1, y1, x2, y2)
```

```
    if m == nil
```

```
        if x1 == x3 and ((y1..y2).cover? y3 or (y2..y1).cover? y3)
```

```
            return true
```

```
        else
```

```
            return false
```

```
        end
```

```
    else
```

```
        n = yAchsenabschnittBestimmen(x1, y1, m)
```

```
        if yDurchXBestimmen(x3, m, n).round(9) == y3.round(9) and ((x1..x2).cover? x3 or (x2..x1).cover? x3)
```

```
            return true
```

```
        else
```

```
            return false
```

```
        end
```

```
    end
```

```
end
```

Diese Methode kontrolliert, ob ein Punkt auf einer Geraden zwischen zwei Punkten liegt. Von den ersten beiden übergebenen Punkten ($x_1|y_1$ und $x_2|y_2$) wird dafür die Steigung bestimmt.

Falls $m = \text{nil}$ gilt wird kontrolliert, ob der x Wert vom dritten Punkt ($x_3|y_3$) der selbe ist wie der vom ersten Punkt und ob die y Koordinate des dritten Punktes innerhalb des Bereiches von y_1 bis y_2 liegt. Ist dies der Fall wird true zurückgegeben, andernfalls false.

Wenn hingegen $m \neq \text{nil}$ gilt, so wird kontrolliert, ob der y Wert der Geraden an der Stelle x_3 dem Wert y_3 entspricht und ob x_3 im Bereich von x_1 bis x_2 liegt. Ist dies der Fall, wird true zurückgegeben, andernfalls false.

Die folgende Methode kontrolliert, ob eine Linie durch ein Polygon hindurch geht. Dies funktioniert nur für den Fall, dass die Linie von einem Punkt des Polygons zu einem anderen Punkt dieses Polygons geht.

```
def punktVonLinieLiegtnInPolygon(knoten, polygonLinien, id, x1, y1, x2, y2)
```

```
    z = 0
    i = 0
    l = knoten.length
    xArr = Array.new
    yArr = Array.new
```

```
    while i < l
        if id == knoten[i][2]
            xArr.push(knoten[i][0])
            yArr.push(knoten[i][1])
        end
        i += 1
    end
```

Zunächst werden alle x und y Koordinaten des Polygons, für welches die Kontrolle stattfindet, herausgesucht und in den Arrays xArr und yArr gespeichert.

```
y = (y1+y2)/2
x = (x1+x2)/2
if y1 == y2
    while not xArr.index(x) == nil
        x = (x+x1)/2
    end

    x3 = x
    y3 = yArr.max + 100
else
    while not yArr.index(y) == nil
        y = (y+y1)/2
        x = (x+x1)/2
    end

    x3 = xArr.max + 100
    y3 = y
end
```

Aus der übergebenen Linie (die zwei Punkte $x_1|y_1$ und $x_2|y_2$) wird anschließend ein Punkt ausgesucht. Dieser Punkt ist zunächst der Mittelpunkt zwischen den übergebenen Punkten. Sollte es jedoch bereits einen Punkt (Knoten) an dieser (x oder y) Stelle geben, wird der Punkt auf den Mittelpunkt von sich selber und dem ersten übergebenen Punkt gesetzt. Dies wiederholt sich solange, bis der Punkt nicht mehr auf einer Höhe mit einem Knoten des Polygons ist.

Dies geschieht, um eine gerade Linie (entweder parallel zur x-Achse oder parallel zur y-Achse) legen zu können, die mit absoluter Sicherheit nicht durch eine Ecke des Polygons verläuft. Auf diese Weise kann dann mit dem folgenden Code Abschnitt der [Punkt-in-Polygon-Test nach Jordan](#) vollzogen werden:

```

i = 0
l = polygonLinien.length
while i < l
    if polygonLinien[i][4] == id
        pX1 = polygonLinien[i][0]
        pY1 = polygonLinien[i][1]
        pX2 = polygonLinien[i][2]
        pY2 = polygonLinien[i][3]

        if punktLiegtAufLinie(pX1, pY1, pX2, pY2, x, y)
            return true
        end

        a = schneidetInBereich(pX1, pY1, pX2, pY2, x, y, x3, y3)
        if a == true
            z += 1
        end
    end

    i += 1
end

return !z.even?
end

```

Es wird im letzten Teil der Methode kontrolliert, wie viele Linien des Polygons die Gerade vom ausgewählten Punkt aus bis zum oben festgelegten Endpunkt (also die Gerade von $x|y$ bis $x3|y3$) schneidet. Ist die Zahl gerade, wird false zurückgegeben, andernfalls true. Liegt der ausgewählte Punkt ($x|y$) zudem auf einer der Linien des Polygons wird sofort true zurückgegeben.

def distanzVonZweiPunkten(x1, y1, x2, y2)

Dem Satz des Pythagoras folgend wird bei dieser Methode der Abstand der übergebenen Punkte $x1|y1$ und $x2|y2$ errechnet und zurückgegeben.

Die folgende Methode ist die mit entscheidendste. Es wird für jeden Knoten (Ecken der Polygone + Lisas Haus) getestet, zu welchem anderen Knoten eine Verbindung ohne Hindernisse besteht. So entsteht ein Weg-Netz durch die Polygone. Zugleich wird auch kontrolliert, von welchem Knoten aus es mit der optimalen Steigung (siehe oben) einen direkten Weg zur y-Achse gibt.

```
def netzLegen(unverbundeneKnoten, polygonLinien)
    i1 = i2 = i3 = 0
    l1 = unverbundeneKnoten.length
    l2 = polygonLinien.length
    knoten = Array.new(l1) {Array.new(l1)}
    zielKnoten = Array.new
    linien = ""
```

Die Methode gibt zum Schluss die Arrays knoten und zielKnoten sowie den String linien zurück. Die Variablen i1, i2, i3, l1 und l2 werden für die nachfolgenden Schleifen verwendet:

```
while i1 < l1
    i2 = 0
    x1, y1 = unverbundeneKnoten[i1][0], unverbundeneKnoten[i1][1]
```

In jedem Durchlauf der ersten while-Schleife wird der nächste noch nicht getestete Knoten genommen und dessen x und y Werte in x1 und y1 gespeichert.

```
while i2 < l1
    i3 = 0
    keineHindernisse = true
    x2, y2 = unverbundeneKnoten[i2][0], unverbundeneKnoten[i2][1]
```

In jedem Durchlauf der zweiten while-Schleife wird der nächste noch nicht getestete Knoten, zu dem der Knoten aus der ersten while-Schleife verbunden werden könnte, genommen und dessen x und y Werte in x2 und y2 gespeichert.

```
if not i1 == i2
```

Mit dieser Verzweigung wird ausgeschlossen, dass ein Knoten sich mit sich selber verbindet.

```
if not sindPunkteNachbarn(polygonLinien, unverbundeneKnoten[i1][2], x1, y1, x2, y2)
```

Wenn die zwei Knoten die gerade in x1|y1 und x2|y2 gespeichert sind zum selben Polygon gehören und benachbart sind, wird der nachfolgende Teil ausgelassen, da diese Knoten - unter der Annahme, dass Polygone weder sich selber noch andere Polygone schneiden, wie es in den Beispielen der Fall ist - zwangsläufig eine Verbindung haben.

```

while i3 < l2 and keineHindernisse
    a = schneidetInBereich(x1, y1, x2, y2, polygonLinien[i3][0],
polygonLinien[i3][1], polygonLinien[i3][2], polygonLinien[i3][3])
    if a == true
        keineHindernisse = false
    end
    i3 += 1
end

```

Die dritte while-Schleife kontrolliert hier, ob die Linie vom Punkt x1|y1 zum Punkt x2|y2 sich mit irgendeiner der Linien der Polygone schneidet.

```

if unverbundeneKnoten[i1][2] == unverbundeneKnoten[i2][2] and
punktVonLinieLiegtnPolygon(unverbundeneKnoten, polygonLinien, unverbundeneKnoten[i1][2], x1, y1, x2,
y2)
    keineHindernisse = false
end

```

Diese if-Verzweigung verhindert, dass eine Linie innerhalb eines Polygons erschaffen wird, was dann passieren kann, wenn zwei Punkte zum selben Polygon gehören aber nicht benachbart sind.

```

end

if keineHindernisse
    linien += "<line stroke='#FF0000' opacity='0.25' x1='#{x1}' y1='#{y1}' x2='#{x2}'"
    y2='#{y2}' stroke-width='1' />"
end

```

Im Falle das die Linie zwischen den Punkten x1|y1 und x2|y2 gelegt werden kann, wird diese Linie als SVG-Linie dem String linien hinzugefügt.

```

knoten[i1][i2] = [keineHindernisse, distanzVonZweiPunkten(x1, y1, x2, y2)]
else
    knoten[i1][i2] = [false]
end

```

Zudem wird in den Array knoten abgespeichert, ob eine Verbindung möglich ist sowie die Distanz zwischen den Knoten x1|y1 und x2|y2

```

i2 += 1
end

```

Anschließend folgt noch innerhalb der ersten while-Schleife eine Kontrolle darauf, ob es sich bei dem derzeitigen Knoten um einen möglichen Ziel-Knoten handelt.

```

keineHindernisse = true

x1, y1, x2 = unverbundeneKnoten[i1][0], unverbundeneKnoten[i1][1], 0
y2 = yAchsenabschnittBestimmen(x1, y1, -((1.0/3)**0.5))
i3 = 0
while i3 < l2 and keineHindernisse
    a = schneidetInBereich(x1, y1, x2, y2, polygonLinien[i3][0], polygonLinien[i3][1],
    polygonLinien[i3][2], polygonLinien[i3][3])
    if a == true
        keineHindernisse = false
    end
    i3 += 1
end

if keineHindernisse
    linien += "<line stroke='#FF0000' opacity='0.25' x1='#{x1}' y1='#{y1}' x2='#{x2}' y2='#{y2}'  

stroke-width='1'/'>"
    zielKnoten.push([y2, i1, distanzVonZweiPunkten(x1, y1, x2, y2)])
end

```

Dies geschieht auf die selbe Art wie es in der dritten while-Schleife weiter oben stattfindet, nur das hier für x2 der Wert 0 eingesetzt ist und für y2 der Wert, den Lisa beim laufen mit optimaler Steigung vom Punkt x1|y1 aus auf der y-Achse erreichen müsste.

```

i1 += 1
end

return linien, knoten, zielKnoten
end

```

Zuletzt wird alles nur noch zurückgegeben.

In der nachfolgenden Methode wird der beste Weg, den Lisa nehmen kann, bestimmt. Dies geschieht durch Anwendung des [Dijkstra-Algorithmus](#).

```

def dijkstra(knoten, unverbundeneKnoten, zielKnoten)
    abstaende = Array.new(knoten.length) {Float::INFINITY}
    abstaende[0] = 0.0
    besterVorherigerKnoten = Array.new(knoten.length) {nil}
    uebrigeKnoten = Array.new(knoten.length) {[|i| i]}

```

Es werden zunächst drei Arrays erschaffen. Der Array abstaende beinhaltet die beste Distanz zu jedem Knoten. Zunächst also für alle Konten bis auf den ersten ∞ und für den ersten Knoten 0. Der Array besterVorherigerKnoten enthält für jeden Knoten den besten Vorgänger Knoten über den er erreicht werden kann. Anfangs deshalb mit nil gefüllt. Der Array uebrigeKnoten beinhaltet alle Knoten die durch die Schleife noch überprüft werden müssen.

```

while uebrigeKnoten.length > 0
    i = 0
    min = Float::INFINITY
    ind = nil
    while i < uebrigeKnoten.length
        if abstaende[uebrigeKnoten[i]] < min
            min = abstaende[uebrigeKnoten[i]]
            ind = uebrigeKnoten[i]
        end
        i += 1
    end
    uebrigeKnoten.delete_at(uebrigeKnoten.index(ind))

```

Zunächst wird der kleinste noch übrige Knoten ausgewählt und aus der Liste der noch übrigen Knoten gelöscht.

```

c = 0
while c < knoten[ind].length
    if knoten[ind][c][0] == true
        if abstaende[c] == Float::INFINITY
            abstaende[c] = abstaende[ind] + knoten[ind][c][1]
            besterVorherigerKnoten[c] = ind
        else
            neuerAbstand = abstaende[ind] + knoten[ind][c][1]
            if abstaende[c] > neuerAbstand
                abstaende[c] = neuerAbstand
                besterVorherigerKnoten[c] = ind
                uebrigeKnoten.push(c)
            end
        end
    end
    c += 1
end

```

Anschließend wird für den ausgewählten Knoten jede Verbindung zu anderen Knoten getestet. Sollte dabei eine Verbindung zu einem anderen Knoten einen kürzeren Weg darstellen, als der bisher gefundene Wert für diesen anderen Knoten, wird der Wert überschrieben und der beste Vorgänger Knoten aktualisiert. In diesem Fall wird der Knoten, zu dem ein besserer Weg gefunden wurde wieder zur Liste der übrigen Knoten hinzugefügt, sodass dessen nachfolgenden Knoten ebenfalls aktualisiert werden.

Im letzten Teil der Methode wird für jeden zuvor beim Netzlegen bestimmten Ziel Knoten die tatsächliche Start und Ziel Zeit berechnet.

```

wege = Array.new
gelbeLinien = ""
v = 0
while v < zielKnoten.length
    grueneLinie = ""
    strecke = abstaende[zielKnoten[v][1]] + zielKnoten[v][2]
    benoetigteZeitBisZielKnoten = (strecke/(15/3.6))
    zeitBisBusAnZielKnotenAnkommt = (zielKnoten[v][0]/(30/3.6))
    weg = Array.new

```

```

zeitAbstand = (zeitBisBusAnZielKnotenAnkommt - benoetigteZeitBisZielKnoten)

index = zielKnoten[v][1]
grueneLinie += "<line stroke='#FFFF00' opacity='1' x1='#{0.0}' y1='#{zielKnoten[v][0]}'"
x2='#{unverbundeneKnoten[index][0]}' y2='#{unverbundeneKnoten[index][1]}' stroke-width='1'/'>"
weg.push([0.0, zielKnoten[v][0], "Treffpunkt mit Bus"])
while not index == 0
    vorherigerIndex = besterVorherigerKnoten[index]
    grueneLinie += "<line stroke='#FFFF00' opacity='1'"
    x1='#{unverbundeneKnoten[index][0]}' y1='#{unverbundeneKnoten[index][1]}'
    x2='#{unverbundeneKnoten[vorherigerIndex][0]}' y2='#{unverbundeneKnoten[vorherigerIndex][1]}'
    stroke-width='1'/'>"
    weg.push([unverbundeneKnoten[index][0], unverbundeneKnoten[index][1],
    "P#{unverbundeneKnoten[index][2]}"])
    index = vorherigerIndex
end
weg.push([unverbundeneKnoten[0][0], unverbundeneKnoten[0][1], "L"])
wege.push([zeitAbstand, grueneLinie, Strecke, benoetigteZeitBisZielKnoten,
zeitBisBusAnZielKnotenAnkommt, weg.reverse])
gelbeLinien += grueneLinie

v += 1
end

```

```

zeitAbstand = wege.max[0].round(0) + 27000
stunden, Minuten, sekunden = zeitInStundenMinutenUndSekunden(zeitAbstand)

puts "Um den Bus noch zu erreichen muss Lisa spätestens um
#{stunden}:#{Minuten}:#{sekunden} los laufen auf der folgenden Strecke:"
i = 0
weg = wege.max[5]
puts "\nx | y | id des Polygons, welches berührt wird"
while i < weg.length
    puts weg[i].join(" | ")

    i += 1
end

zeitBisBusAnZielKnotenAnkommt = wege.max[4]
stunden, Minuten, sekunden =
zeitInStundenMinutenUndSekunden(zeitBisBusAnZielKnotenAnkommt.round(0)+27000)
puts "\nLisa erreicht auf diesem Weg den Bus um #{stunden}:#{Minuten}:#{sekunden} - der
Weg hat eine Gesamtlänge von #{wege.max[2]} Metern."
benoetigteZeitBisZielKnoten = wege.max[3]
stunden, Minuten, sekunden =
zeitInStundenMinutenUndSekunden(benoetigteZeitBisZielKnoten.round(0))
puts "\nSie benötigt dafür eine Zeit von #{stunden}h #{Minuten}min #{sekunden}sek"

return gelbeLinien + wege.max[1].gsub!("#FFFF00", "#00FF00")
end

```

Es werden alle besten Wege zu den Ziel Knoten als gelbe SVG Linien gespeichert. Der Ziel Knoten mit der besten Start Zeit (also dem höchsten Wert in der Variable zeitAbstand) wird ausgewählt und der Weg sowie Start und Ziel (Ankunft am Bus) Zeit von diesem werden ausgegeben. Zudem wird der beste Weg als grüne SVG Linie gespeichert.

```
def zeitInStundenMinutenUndSekunden(sekunden)
```

Die Methode erhält eine Anzahl an Sekunden und gibt diese in Stunden, Minuten und Sekunden zurück.

```
puts "Gebe den Namen der einzulesenen .txt Datei ein:"  
unverbundeneKnoten, polygonLinien = polygoneEinlesen(gets.chomp)  
svgLinien, knoten, zielKnoten = netzLegen(unverbundeneKnoten, polygonLinien)  
  
svgLinien += dijkstra(knoten, unverbundeneKnoten, zielKnoten)  
startPunkt = "<circle xmlns='http://www.w3.org/2000/svg' id='L' cx='#{unverbundeneKnoten[0][0]}' cy='#{unverbundeneKnoten[0][1]}' r='10' fill='#F42121' stroke='#000080' stroke-width='1' />"  
newSVGName = "#{Time.now.to_s[0..-7].gsub!(:, "-")}.svg"  
puts "\nLisas Strecke wurde in folgender Datei abgespeichert: #{newSVGName}\nDie roten Linien stellen alle Wege zwischen den Ecken der Polygone dar, die gelben Linien alle direkten Wege zum Rand und die grüne Linie zeigt den besten gefunden Weg an, den Lisa laufen kann."  
newSVG = File.new(newSVGName, "w+")  
newSVG.write("<svg version='1.1' viewBox='0 0 1024 1400' xmlns='http://www.w3.org/2000/svg'><g transform='scale(1 -1)'><g transform='translate(0 -1200)'><line xmlns='http://www.w3.org/2000/svg' id='y' x1='0' x2='0' y1='0' y2='1200' fill='none' stroke='#212121' stroke-width='3' />#$svgPolygone}#{startPunkt}#{svgLinien}</g></g></svg>")  
newSVG.close
```

Zum Schluss folgt die Dateiabfrage und das Aufrufen der verschiedenen Methoden. Anschließend werden alle zwischengespeicherten SVG Abschnitte geordnet zusammen gesetzt und abgespeichert. Der Name der dabei entstehenden SVG Datei beinhaltet das Erstellungsdatum sowie der Erstellungszeit.

Beispiele

```
C:\Windows\system32\cmd.exe
I:\BWINF.2\lisa_rennnt>lisa_rennnt.rb
Geben den Namen der einzulesenen .txt Datei ein:
lisarennnt1.txt
Um den Bus noch zu erreichen muss Lisa spätestens um 7:28:0 los laufen auf der folgenden Strecke:
x | y | id des Polygons, welches berührt wird
633.0 | 189.0 | L
535.0 | 410.0 | P1
9.0 | 718.8823940164498 | Treffpunkt mit Bus

Lisa erreicht auf diesem Weg den Bus um 7:31:26 - der Weg hat eine Gesamtlänge von 859.5187952385713 Metern.

Sie benötigt dafür eine Zeit von 0h 3min 26sek.

Lisas Strecke wurde in folgender Datei abgespeichert: 2019-04-28 13-46-33.svg
Die roten Linien stellen alle Wege zwischen den Ecken der Polygone dar, die gelben Linien alle direkten Wege zum Rand und die grüne Linie zeigt den besten gefunden Weg an, den Lisa laufen kann.

I:\BWINF.2\lisa_rennnt>lisa_rennnt.rb
Geben den Namen der einzulesenen .txt Datei ein:
lisarennnt2.txt
Um den Bus noch zu erreichen muss Lisa spätestens um 7:28:9 los laufen auf der folgenden Strecke:
x | y | id des Polygons, welches berührt wird
633.0 | 189.0 | L
505.0 | 213.0 | P1
390.0 | 260.0 | P1
170.0 | 402.0 | P3
0.0 | 500.14954576223636 | Treffpunkt mit Bus

Lisa erreicht auf diesem Weg den Bus um 7:31:0 - der Weg hat eine Gesamtlänge von 712.6105908652551 Metern.

Sie benötigt dafür eine Zeit von 0h 2min 51sek.

Lisas Strecke wurde in folgender Datei abgespeichert: 2019-04-28 13-46-39.svg
Die roten Linien stellen alle Wege zwischen den Ecken der Polygone dar, die gelben Linien alle direkten Wege zum Rand und die grüne Linie zeigt den besten gefunden Weg an, den Lisa laufen kann.

I:\BWINF.2\lisa_rennnt>lisa_rennnt.rb
Geben den Namen der einzulesenen .txt Datei ein:
lisarennnt3.txt
Um den Bus noch zu erreichen muss Lisa spätestens um 7:27:29 los laufen auf der folgenden Strecke:
x | y | id des Polygons, welches berührt wird
575.0 | 158.0 | L
519.0 | 238.0 | P2
520.0 | 250.0 | P2
490.0 | 298.0 | P3
426.0 | 338.0 | P8
390.0 | 288.0 | P5
352.0 | 287.0 | P6
291.0 | 296.0 | P6
0.0 | 464.00892833418106 | Treffpunkt mit Bus

Lisa erreicht auf diesem Weg den Bus um 7:30:56 - der Weg hat eine Gesamtlänge von 862.5844316766388 Metern.

Sie benötigt dafür eine Zeit von 0h 3min 27sek.

Lisas Strecke wurde in folgender Datei abgespeichert: 2019-04-28 13-46-46.svg
Die roten Linien stellen alle Wege zwischen den Ecken der Polygone dar, die gelben Linien alle direkten Wege zum Rand und die grüne Linie zeigt den besten gefunden Weg an, den Lisa laufen kann.

I:\BWINF.2\lisa_rennnt>lisa_rennnt.rb
Geben den Namen der einzulesenen .txt Datei ein:
lisarennnt4.txt
Um den Bus noch zu erreichen muss Lisa spätestens um 7:26:56 los laufen auf der folgenden Strecke:
x | y | id des Polygons, welches berührt wird
856.0 | 270.0 | L
900.0 | 300.0 | P11
900.0 | 340.0 | P11
896.0 | 475.0 | P10
0.0 | 992.3058411939046 | Treffpunkt mit Bus

Lisa erreicht auf diesem Weg den Bus um 7:31:59 - der Weg hat eine Gesamtlänge von 1262.9250364694046 Metern.

Sie benötigt dafür eine Zeit von 0h 5min 3sek.

Lisas Strecke wurde in folgender Datei abgespeichert: 2019-04-28 13-46-54.svg
Die roten Linien stellen alle Wege zwischen den Ecken der Polygone dar, die gelben Linien alle direkten Wege zum Rand und die grüne Linie zeigt den besten gefunden Weg an, den Lisa laufen kann.

C:\Windows\system32\cmd.exe
I:\BWINF.2\lisa_rennnt>lisa_rennnt.rb
Geben den Namen der einzulesenen .txt Datei ein:
lisarennnt5.txt
Um den Bus noch zu erreichen muss Lisa spätestens um 7:27:55 los laufen auf der folgenden Strecke:
x | y | id des Polygons, welches berührt wird
621.0 | 162.0 | L
410.0 | 175.0 | P8
380.0 | 165.0 | P3
326.0 | 165.0 | P3
280.0 | 215.0 | P5
208.0 | 215.0 | P5
176.0 | 165.0 | P6
136.0 | 265.0 | P9
0.0 | 340.05553499465134 | Treffpunkt mit Bus

Lisa erreicht auf diesem Weg den Bus um 7:30:41 - der Weg hat eine Gesamtlänge von 691.1964259470017 Metern.

Sie benötigt dafür eine Zeit von 0h 2min 46sek.

Lisas Strecke wurde in folgender Datei abgespeichert: 2019-04-28 13-47-01.svg
Die roten Linien stellen alle Wege zwischen den Ecken der Polygone dar, die gelben Linien alle direkten Wege zum Rand und die grüne Linie zeigt den besten gefunden Weg an, den Lisa laufen kann.
```

Im Unterordner „Aufgabe 1\Beispiele“ befindet sich die oben zu sehende Ausgabe aus der Konsole für alle fünf Beispiele auf der Webseite sowie die fünf dazugehörigen SVG Dateien.