

# Aufgabe 1: Lisa rennt

Teilnahme ID: 48313

Nikolas Kilian

25. April 2019

## Inhaltsverzeichnis

<b>1</b>	<b>Lösungsidee</b>	<b>1</b>
1.1	Berechnung . . . . .	2
<b>2</b>	<b>Umsetzung</b>	<b>3</b>
<b>3</b>	<b>Beispiele</b>	<b>5</b>
3.1	Beispiel 1 . . . . .	5
3.2	Beispiel 2 . . . . .	6
3.3	Beispiel 3 . . . . .	7
3.4	Beispiel 4 . . . . .	8
3.5	Beispiel 5 . . . . .	9
3.6	Beispiel 6 . . . . .	10
<b>4</b>	<b>Code</b>	<b>11</b>

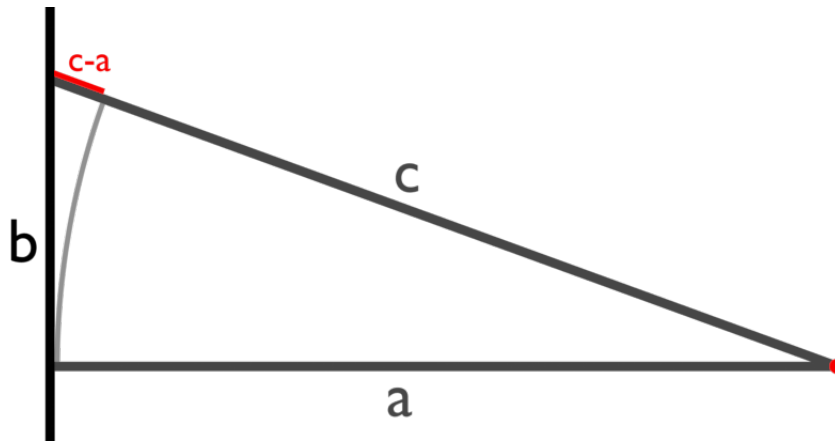
## 1 Lösungsidee

Wenn es keine Hindernisse gibt, so ist der optimale Weg eine gerade Strecke vom Startpunkt zum Buspfad im  $30^\circ$  Winkel. Für Begründung davon siehe 1.1.

Gibt es Hindernisse, so ist der optimale Weg der optimale Weg zu einem Eckpunkt, von dem die  $30^\circ$  Strecke offen ist, und dann diese  $30^\circ$  Strecke.

Um das Optimum mit Hindernissen zu finden, muss man also alle Eckpunkte bestimmen, von denen aus diese  $30^\circ$  Strecke offen ist, und den optimalen Weg zu ihnen bestimmen. Da der optimale Weg das Format der resultierenden Wege hat, ist unter den resultierenden Wegen das Optimum enthalten, also muss man nun nur noch die Zeit, zu der Lisa loslaufen muss, für alle Wege errechnen und den Weg mit der spätesten Startzeit auswählen. Der optimale Weg zu diesen Eckpunkten lässt sich bestimmen mithilfe eines Sichtbarkeitsgraphen und Dijkstra's Algorithmus. Zum verhindern von Strecken durch unendlich dünne Wege (berührende Polygone) verändert man den Sichtbarkeitsgraphen, sodass für jede normal sichtbare Linie nachträglich auf unendlich dünne Wege geprüft werden.

## 1.1 Berechnung



$$c = \sqrt{a^2 + b^2}$$

$$f := \frac{b}{a}$$

Der Zeitvorteil durch eine angewinkelte Strecke ist die Differenz zwischen der Zeit die Lisa braucht für ihre Extrastrecke, und die Zeit die der Bus mehr fährt.

$$\begin{aligned} t(f) &= \frac{c-a}{v_{Lisa}} - \frac{b}{v_{Bus}} \\ &= \frac{\sqrt{a^2 + b^2} - a}{v_{Lisa}} - \frac{af}{v_{Bus}} \\ &= \frac{\sqrt{a^2(1+f^2)} - a}{v_{Lisa}} - \frac{af}{v_{Bus}} \\ &= a \left( \frac{\sqrt{1+f^2} - 1}{v_{Lisa}} - \frac{f}{v_{Bus}} \right) \end{aligned}$$

$$\begin{aligned} \frac{dt(f)}{df} &= \frac{da \left( \frac{\sqrt{1+f^2}-1}{v_{Lisa}} - \frac{f}{v_{Bus}} \right)}{df} \\ &= a \left( \frac{d \frac{\sqrt{1+f^2}-1}{v_{Lisa}}}{df} - \frac{d \frac{f}{v_{Bus}}}{df} \right) \\ &= a \left( \frac{d \frac{\sqrt{1+f^2}}{df}}{v_{Lisa}} - \frac{\frac{df}{df}}{v_{Bus}} \right) \\ &= a \left( \frac{\frac{1}{2\sqrt{1+f^2}} \cdot \frac{d1+f^2}{df}}{v_{Lisa}} - \frac{1}{v_{Bus}} \right) \\ &= a \left( \frac{f}{v_{Lisa}\sqrt{1+f^2}} - \frac{1}{v_{Bus}} \right) \end{aligned}$$

Extremstellen dieser Zeitdifferenz stellen die besten und schlechtesten Winkel für Lisas Strecke da.

$$\begin{aligned}
 & \frac{dt(f_0)}{df} = 0 \\
 \Leftrightarrow & a \left( \frac{f_0}{v_{Lisa} \sqrt{1 + f_0^2}} - \frac{1}{v_{Bus}} \right) = 0 \\
 \Leftrightarrow & a \frac{f_0}{v_{Lisa} \sqrt{1 + f_0^2}} = a \frac{1}{v_{Bus}} \\
 \Leftrightarrow & \frac{f_0}{\sqrt{1 + f_0^2}} = \frac{v_{Lisa}}{v_{Bus}} \\
 \Leftrightarrow & \left( \frac{f_0}{\sqrt{1 + f_0^2}} \right)^2 = \left( \frac{v_{Lisa}}{v_{Bus}} \right)^2 \\
 \Leftrightarrow & \frac{f_0^2}{1 + f_0^2} = \frac{v_{Lisa}^2}{v_{Bus}^2} \\
 \Leftrightarrow & \frac{1 + f_0^2}{f_0^2} = \frac{v_{Bus}^2}{v_{Lisa}^2} \\
 \Leftrightarrow & \frac{1}{f_0^2} = \frac{v_{Bus}^2 - v_{Lisa}^2}{v_{Lisa}^2} \\
 \Leftrightarrow & f_0^2 = \frac{v_{Lisa}^2}{v_{Bus}^2 - v_{Lisa}^2} \\
 \Leftrightarrow & f_0 = \sqrt{\frac{v_{Lisa}^2}{v_{Bus}^2 - v_{Lisa}^2}} \\
 \Leftrightarrow & f_0 = \frac{v_{Lisa}}{\sqrt{v_{Bus}^2 - v_{Lisa}^2}}
 \end{aligned}$$

Für die Standardwerte von Lisas Geschwindigkeit und der Busgeschwindigkeit, ist die Extremstelle  $f_0 = \frac{1}{\sqrt{3}}$ , wobei  $\arctan(f_0) = 30^\circ$ . Somit ist die optimale Strecke im  $30^\circ$  Winkel.

## 2 Umsetzung

Zur Umsetzung habe ich mich für eine Implementation in C# entschieden, mit einer Visualisierung mithilfe von WPF. Für die Generierung von Sichtbarkeitspolygonen verwende ich eine Implementation des Sweep-Line Algorithmus [Sources here]. Die Version des Algorithmus die ich verwende funktioniert wie folgt:

```

1 Let Intersections = Binary Search Tree, sorted by the order of
  intersection
2
3 foreach (Point p in Points sorted by their angle to Origin) {
4     Intersections.RemoveAll(Connected Edges on Clockwise Side of p);
5
6     if (IsVisible(p)) VisibleVertices.Add(p);
7

```

```
8   Intersections.AddAll(Connected Edges on Counterclockwise Side of p)
9   ;
10  }
11  boolean IsVisible(p) {
12      if (!Origin.BetweenNeighbours(p) || !p.BetweenNeighbours(Origin))
13          return false;
14      if (Origin and p are neighbours) return true;
15      if (Intersections is not empty and its leftmost element
16          intersects the line from Origin to Target) return false;
17  }
```

P.BetweenNeighbours(A) gibt dabei zurück, ob für einen Punkt P der Teil eines Polygons ist ob A in dem in Abb. 1 grün markiertem Bereich liegt. Ist das Polygon in P nicht konvex, so ist das Ergebnis immer false.

Wenn der Rückgabewert dieser Methode false ist, so sind in einem reduzierten Sichtbarkeitsgraph die beiden Punkte nicht verbunden.

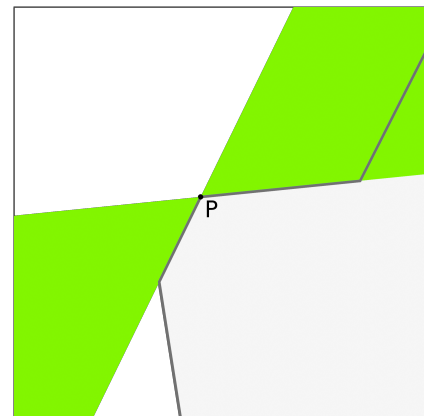


Abbildung 1: BetweenNeighbours

Um das durchgehen unendlich dünner Wege zu verhindern, speichere ich die hinzugefügten/entfernten Kanten, errechne die Strecke die sie auf der Strecke zum aktuellen Punkt einnehmen, und errechne die Überschneidungen der linken und rechten Seite.

Um nun ein reduzierten Sichtbarkeitsgraphen zu generieren muss dieser Algorithmus nun nur noch für alle Punkte ausgeführt werden.

Mit dem Sichtbarkeitsgraphen fertig generiere ich nun eine Heuristik mit Dijkstras Algorithmus, jedoch generiere ich diese nur bis allen Endpunkten (Enden der 30° Strecken, auf dem Buspfad) von Dijkstra besucht wurden (/an der Spitze der Prioritätsliste waren).

Da Dijkstra's Algorithmus nicht immer alle Knoten besucht, muss der Sichtbarkeitsgraph auch nicht vollständig generiert werden. Um dies auszunutzen berechne ich das Sichtbarkeitspolygon nur für Punkte die Dijkstra besucht.

Sobald die Heuristik fertig generiert ist, errechne mit dieser die optimale Strecke zu allen Endpunkten und die Zeit die Lisa braucht um diese abzulaufen, und die Zeit die der Bus braucht, um dorthin zu kommen. Damit errechne ich die Zeit zu der Lisa losgehen muss für alle diese Wege, vergleiche diese und nehme den Weg mit der spätesten Startzeit. Dieser Weg ist der optimale Weg, und somit das Ergebnis.

## 3 Beispiele

### 3.1 Beispiel 1

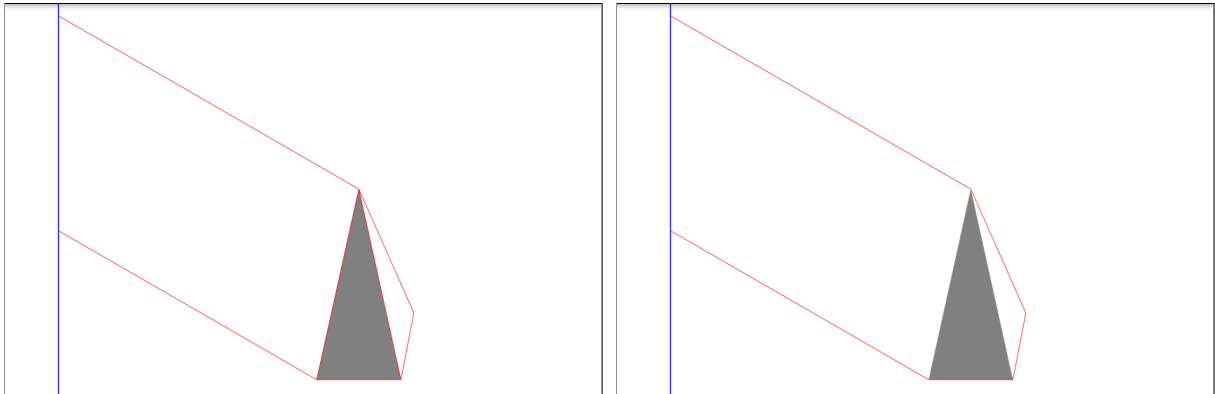


Abbildung 2: Sichtbarkeitsgraph

Abbildung 3: Dijkstra Heuristik

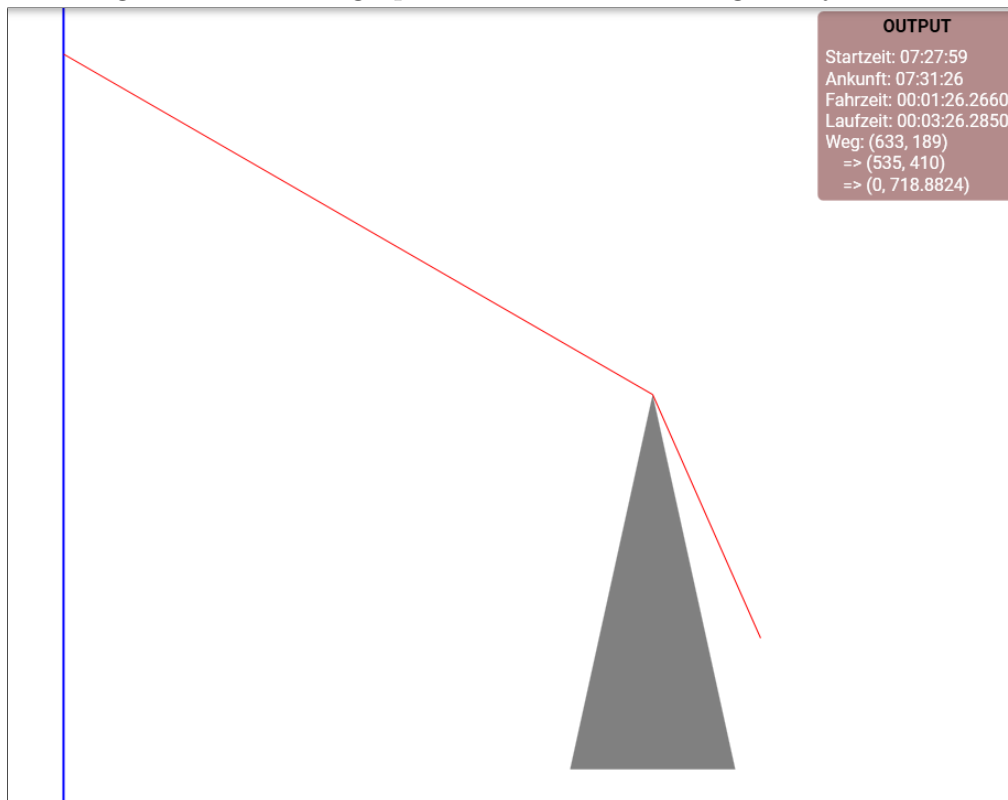


Abbildung 4: optimaler Weg

### 3.2 Beispiel 2

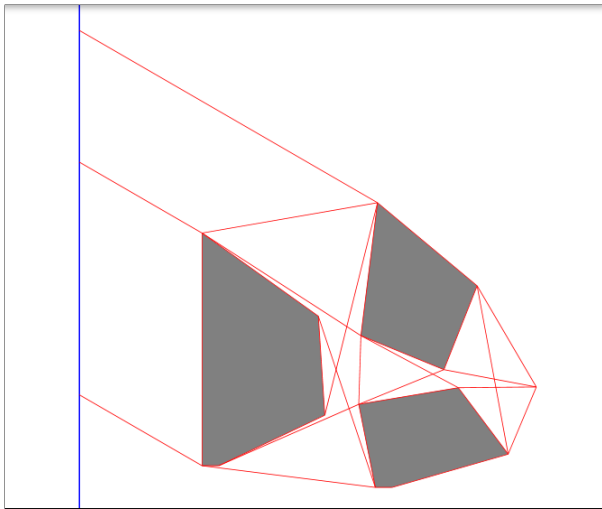


Abbildung 5: Sichtbarkeitsgraph

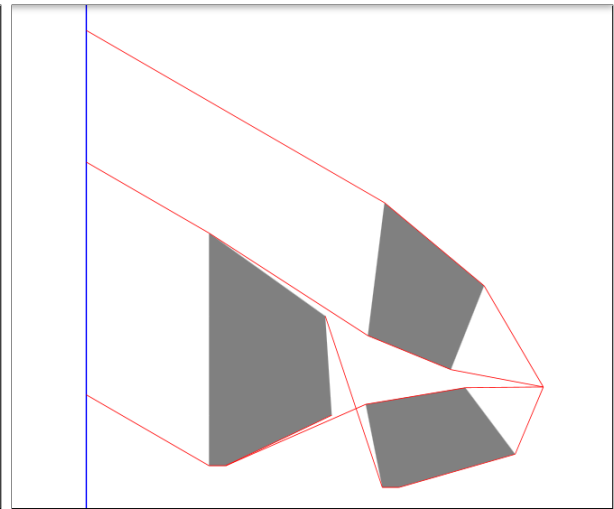


Abbildung 6: Dijkstra Heuristik

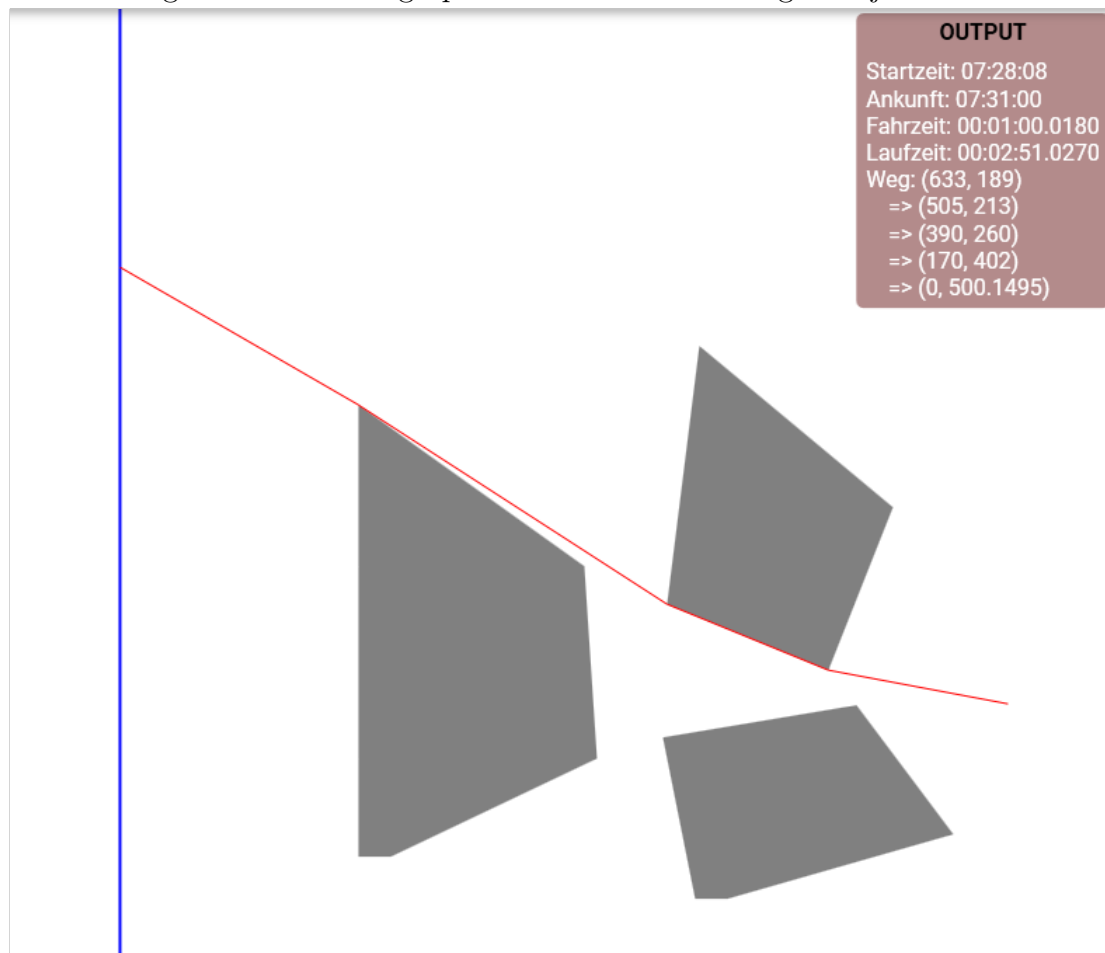


Abbildung 7: optimaler Weg

### 3.3 Beispiel 3

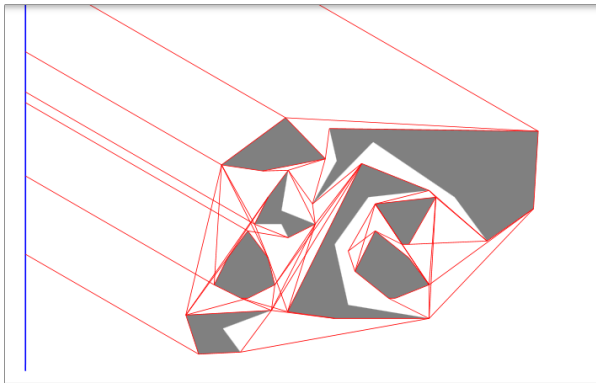


Abbildung 8: Sichtbarkeitsgraph

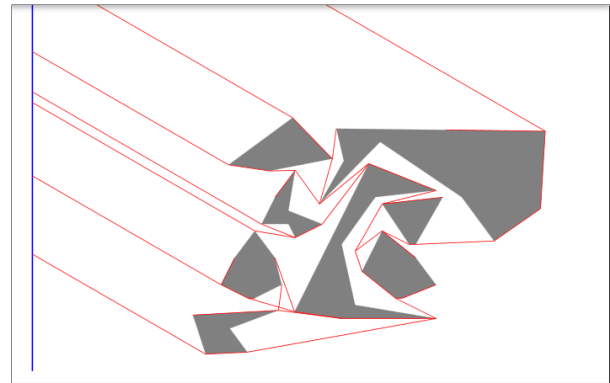


Abbildung 9: Dijkstra Heuristik

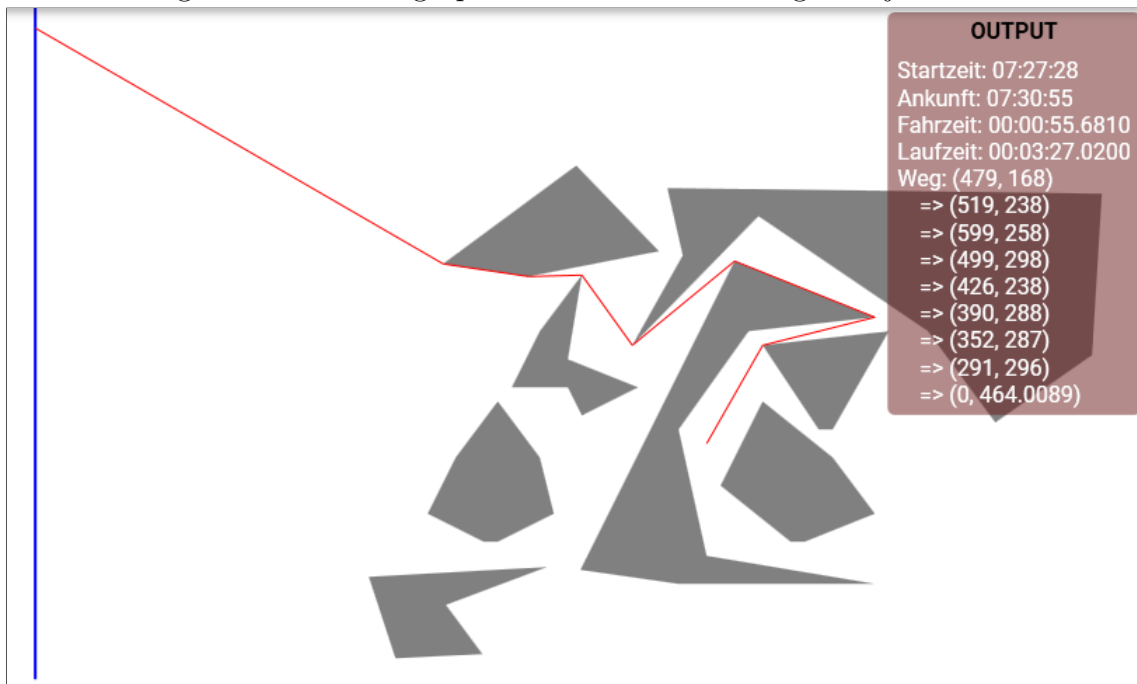


Abbildung 10: optimaler Weg

### 3.4 Beispiel 4

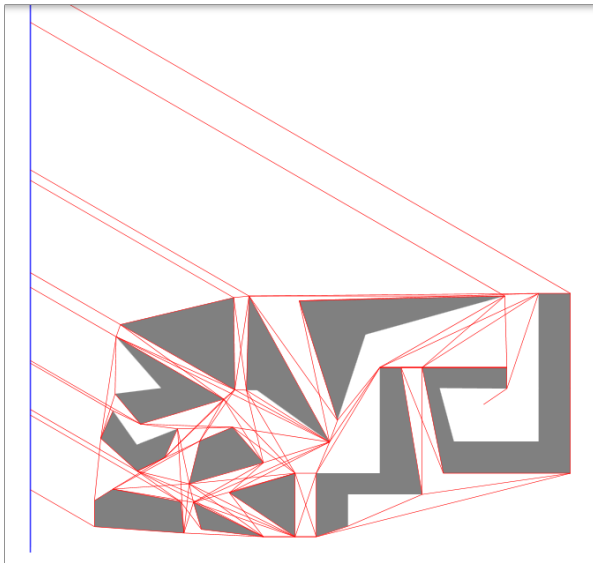


Abbildung 11: Sichtbarkeitsgraph

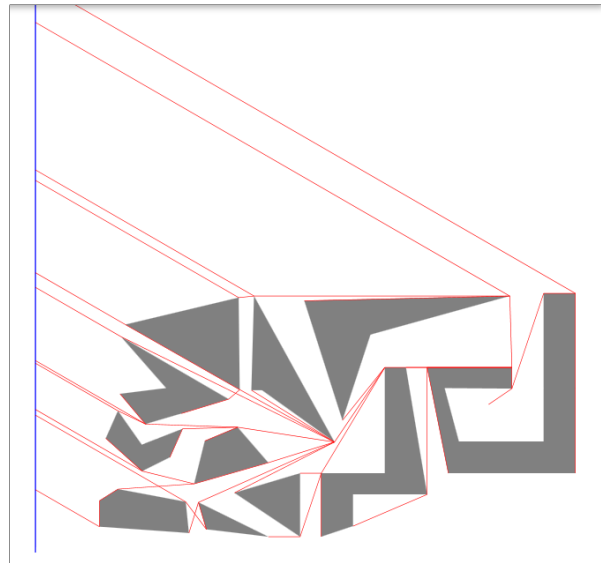


Abbildung 12: Dijkstra Heuristik

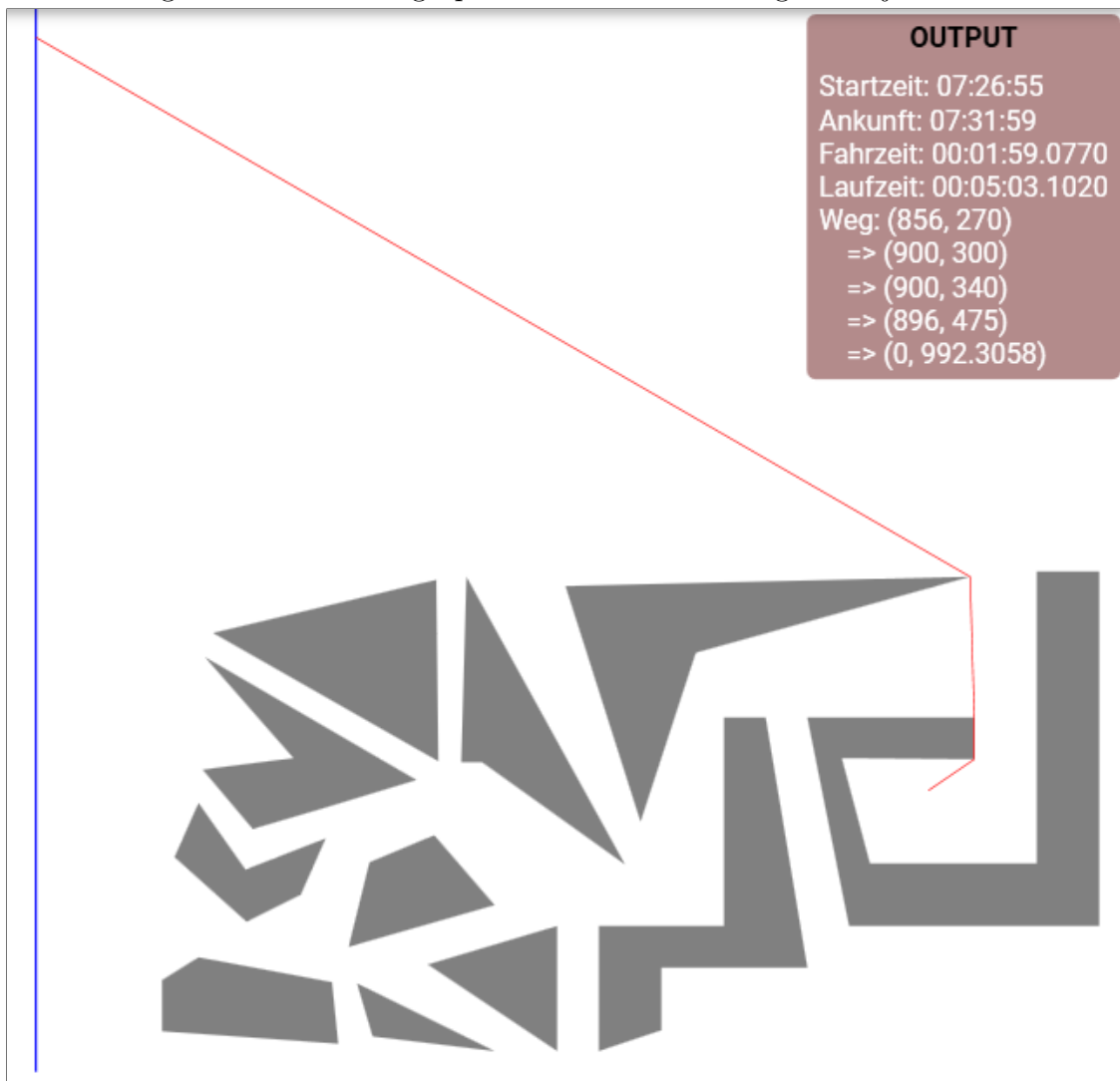


Abbildung 13: optimaler Weg



### 3.5 Beispiel 5

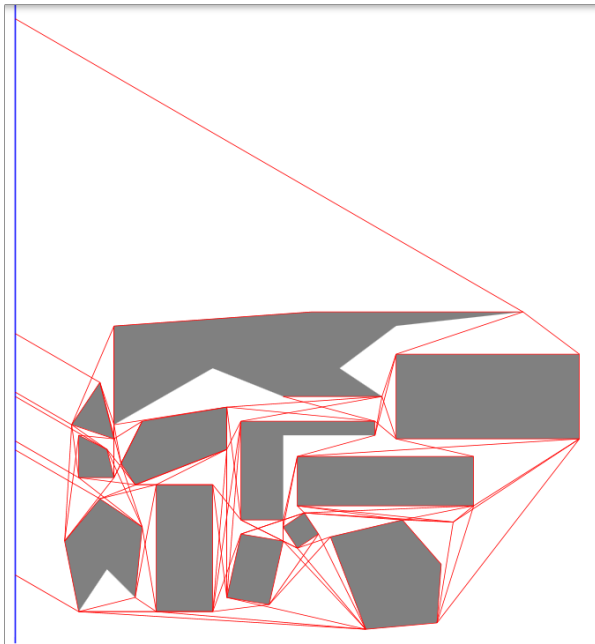


Abbildung 14: Sichtbarkeitsgraph

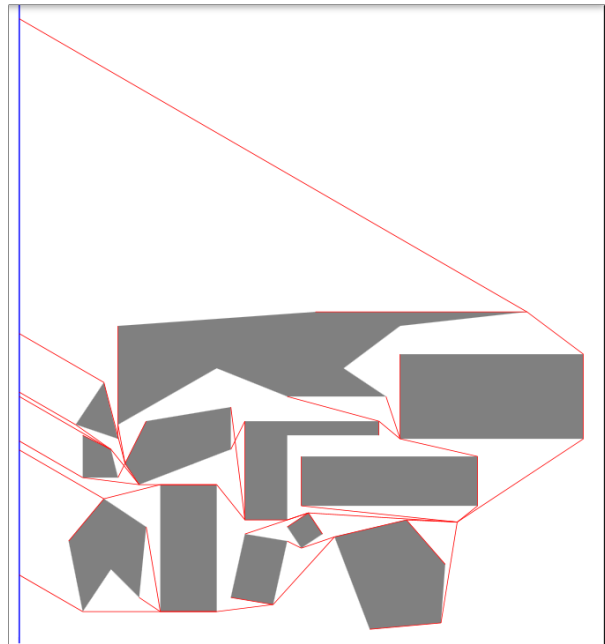


Abbildung 15: Dijkstra Heuristik

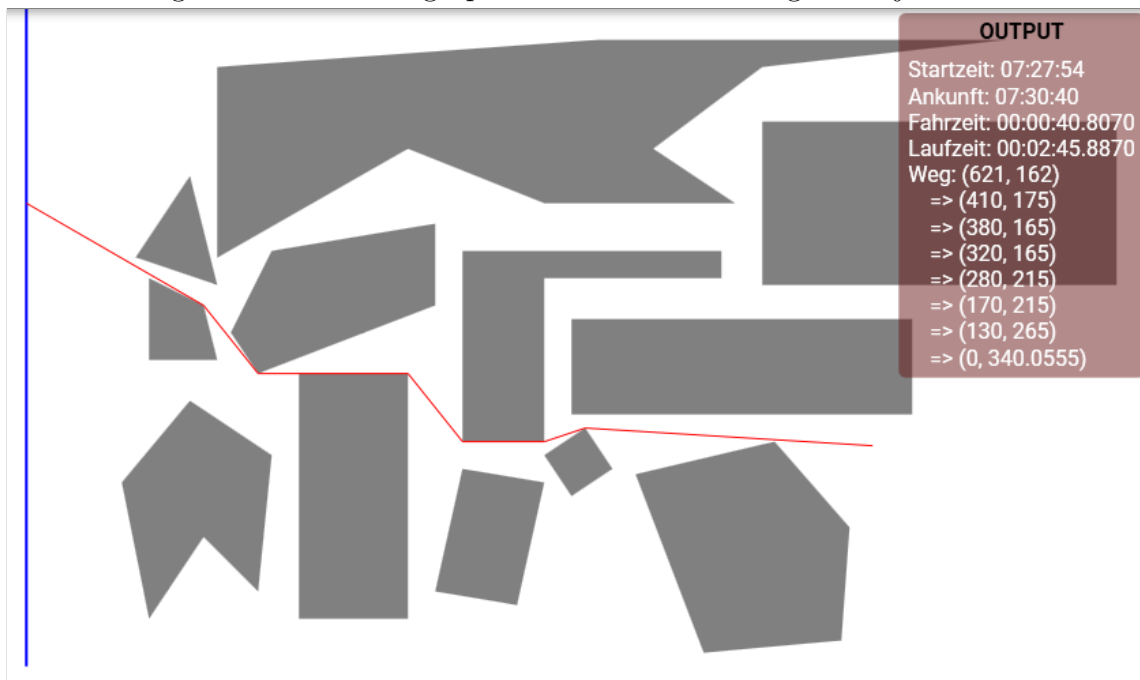


Abbildung 16: optimaler Weg

### 3.6 Beispiel 6

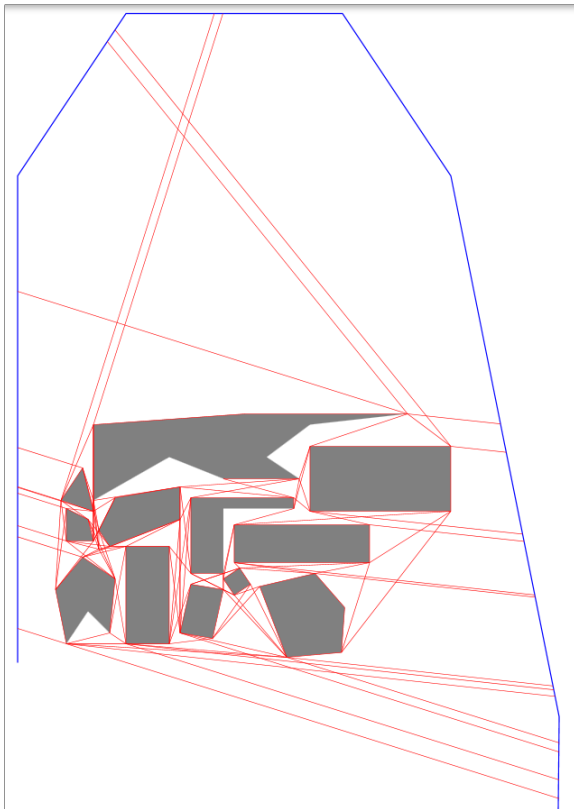


Abbildung 17: Sichtbarkeitsgraph

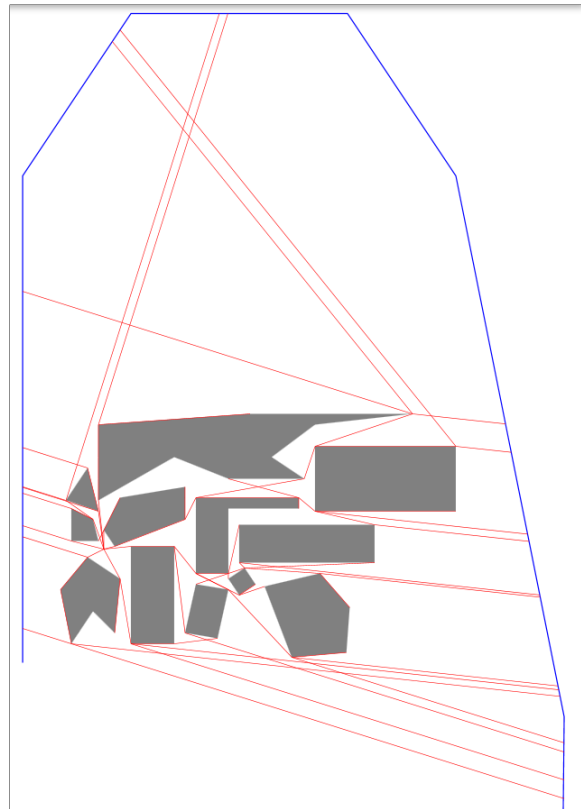


Abbildung 18: Dijkstra Heuristik

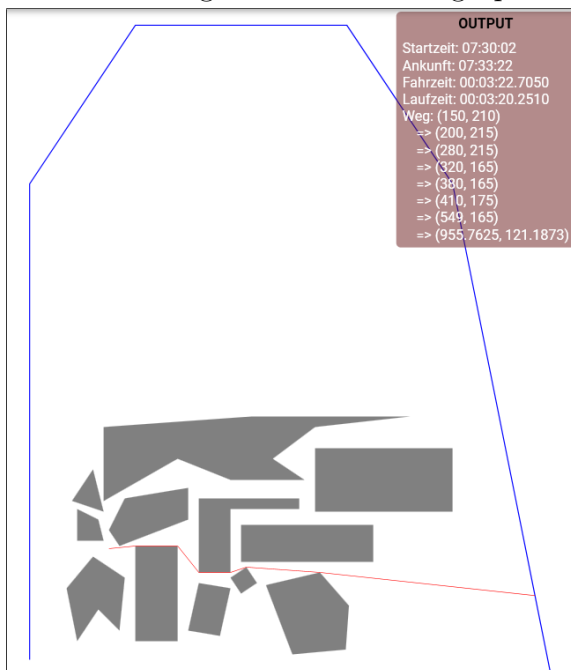


Abbildung 19: optimaler Weg

Dieses Beispiel ist eine Modifikation von Beispiel 5, bei der der Buspfad um die Polygone herumgeht, und der Bus statt mit 30km/h mit 50km/h fährt. In der Datei:

```
1 // Normale Datei wie vorgegeben
2 // Buspfad angegeben wie
  Polygone, wobei inf und -inf
  positive und negative
  Unendlichkeit angeben
3 /*Lisas Geschwindigkeit in km/h
  */ /*Busgeschwindigkeit in
  km/h*/
```

## 4 Code

Listing 1: Results

```
1 List<Vertex> optimalPath = map.GetOptimalPath(out double
    characterLength, out double busLength, out double advantage, out
    var debug);
2
3 DateTime start = DateTime.Now.Let(x => new DateTime(x.Year, x.Month,
    x.Day, 7, 30, 0));
4
5 output.Text =
6 @$"Startzeit:{(start - TimeSpan.FromSeconds(advantage))}
7 Ankunft:{(start + TimeSpan.FromSeconds(busLength / map.busSpeed))}
8 Fahrzeit:{TimeSpan.FromSeconds(busLength / map.busSpeed)}
9 Laufzeit:{TimeSpan.FromSeconds(characterLength / map.characterSpeed)}
10 Weg:{string.Join("=>", Enumerable.Reverse(optimalPath).Select(x => $"
    ({x.vector.x}, {x.vector.y}")))}";
```

Listing 2: class Polygon

```
1 public Vertex[] vertices;
2 public int Length => vertices.Length;
3 public Vertex this[int index] => vertices[MathHelper.PositiveModulo(
    index, 0, Length)];
4
5 // Polygons are always sorted to be Counterclockwise
```

Listing 3: class Vertex

```
1 public readonly Vector vector;
2 public readonly Polygon polygon;
3 public readonly int index;
4
5 public bool notConvex;
6
7 public Vertex Init()
8 {
9     notConvex = Vector.Orientation(Previous.vector, vector, Next.
    vector) == Vector.VectorOrder.Clockwise; // Polygons are
    Counterclockwise
10     return this;
11 }
12
13 public Vertex Previous => polygon[index - 1];
14 public Vertex Next => polygon[index + 1];
15
16 public bool IsNeighbor(Vertex other) => Previous == other || Next ==
    other;
17
18 // Assumes notConvex to be properly calculated
19 public bool BetweenNeighbors(Vector other) =>
20     !notConvex && (Vector.Orientation(vector, Previous.vector, other)
    != Vector.Orientation(Next.vector, vector, other));
```

Listing 4: class Vector

```
1 public double x, y;
2
3 public Vector() { }
4
5 public Vector(double angle) : this(Math.Cos(angle), Math.Sin(angle))
    { }
6
7 public Vector(double x, double y)
8 {
9     this.x = x;
10    this.y = y;
11 }
12
13 public Vector Left => new Vector(-y, x);
14 public Vector Right => new Vector(y, -x);
15 public Vector Back => new Vector(-x, -y);
16
17 // Algorithm from https://bryceboe.com/2006/10/23/line-segment-
    intersection-algorithm/
18 public enum VectorOrder : int
19 {
20     Collinear = -1,
21     Clockwise = 0,
22     Counterclockwise = 1,
23 }
24 /// <summary>
25 /// Calculates the orientation of the triangle defined by a, b and c
26 /// </summary>
27 public static VectorOrder Orientation(Vector a, Vector b, Vector c)
28 {
29     double orientation = (c.y - a.y) * (b.x - a.x) - (b.y - a.y) * (c
        .x - a.x);
30
31     if (orientation < 0) return VectorOrder.Clockwise;
32     if (orientation == 0) return VectorOrder.Collinear;
33     if (orientation > 0) return VectorOrder.Counterclockwise;
34
35     throw new NotFiniteNumberException();
36 }
37 /// <summary>
38 /// Like Orientation, but provides a margin for collinearity
39 /// </summary>
40 /// <param name="epsilon">The margin for collinearity</param>
41 public static VectorOrder OrientationApprox(Vector a, Vector b,
    Vector c, double epsilon)
42 {
43     double orientation = (c.y - a.y) * (b.x - a.x) - (b.y - a.y) * (c
        .x - a.x);
44
45     if (orientation < -epsilon) return VectorOrder.Clockwise;
46     if (orientation > epsilon) return VectorOrder.Counterclockwise;
47
48     if (double.IsNaN(orientation)) throw new NotFiniteNumberException
        ();
49     return VectorOrder.Collinear;
50 }
```

```

51 public static bool IntersectingLines(Vector startA, Vector endA,
    Vector startB, Vector endB)
52 {
53     VectorOrder sAsBeB = Orientation(startA, startB, endB);
54     VectorOrder eAsBeB = Orientation(endA, startB, endB);
55     VectorOrder sAeAsB = Orientation(startA, endA, startB);
56     VectorOrder sAeAeB = Orientation(startA, endA, endB);
57
58     return (sAsBeB != eAsBeB && sAeAsB != sAeAeB)
59         && !(sAeAeB == VectorOrder.Collinear || eAsBeB ==
            VectorOrder.Collinear || sAeAsB == VectorOrder.
            Collinear || sAeAeB == VectorOrder.Collinear);
60 }
    
```

#### Listing 5: class Helper

```

1 public static int PositiveModulo(int value, int offset, int length)
2 {
3     while (value < offset) value += length;
4     while (value >= offset + length) value -= length;
5     return value;
6 }
7 public static double PositiveModulo(double value, double offset,
    double length)
8 {
9     while (value < offset) value += length;
10    while (value >= offset + length) value -= length;
11    return value;
12 }
13 public static double ModuloAngle(double angle) => PositiveModulo(
    angle, 0, 2 * Math.PI);
14
15 public static double Clamp(double value, double min, double max) =>
    value < min ? min : value > max ? max : value;
16
17 public static bool Approx(this double first, double second, double
    epsilon) => Math.Abs(first - second) < epsilon;
18 public static bool Approx(this Vector first, Vector second, double
    epsilonSquared) => first.DistanceSquared(second) < epsilonSquared;
19
20 public static (T1 value, T2 comparable) MinValue<T1, T2>(this
    IEnumerable<T1> enumerable, Func<T1, T2> selector) where T2 :
    IComparable<T2>
21 // Returns the element with the lowest return value selector(x) out
    of an enumerable, along with that return value
22 public static T MaxValue<T>(this IEnumerable<T> enumerable,
    Comparison<T> comparer)
23 // Returns the element with the highest element out of an enumerable,
    as defined by the given comparer
24
25 public static T2 Let<T1, T2>(this T1 obj, Func<T1, T2> func) => func(
    obj);
26 public static void Let<T>(this T obj, Action<T> action) => action(obj
    );
    
```

#### Listing 6: class Map

```
1 public Polygon[] polygons;
2 public Vector[] busPath;
3 public Vertex startingPosition;
4 public List<Vertex> allPolygonVertices;
5 public double busSpeed, characterSpeed, busApproachConstant;
6
7 public void SetSpeed(double characterSpeed, double busSpeed)
8 {
9     this.characterSpeed = characterSpeed;
10    this.busSpeed = busSpeed;
11    busApproachConstant = characterSpeed / Math.Sqrt(busSpeed *
12        busSpeed - characterSpeed * characterSpeed);
13 }
14 public IEnumerable<Vector> GetEndpoints(Vector dot)
15 // Returns all endpoints of the direct paths (30 degree angle) from
16 // the given Vector to the bus path
17 public double CalculateDistanceAtAngle(Vertex vertex, Vector origin,
18     double angle)
19 // Calculates the distance from origin to the intersection between a
20 // ray from origin with a given angle, and the line containing vertex
21 // and vertex.Next
22 public double epsilon = 1E-15;
23
24 public List<Vertex> GenerateVisibilityPolygon(Vertex originVertex,
25     out List<(Vertex vert, double busLength)> endpoints, out List<(
26     Vector, Vector)> debug)
27 {
28     List<(Vector, Vector)> debugOut = new List<(Vector, Vector)>();
29
30     Vector origin = originVertex.vector;
31
32     List<Vertex> visibilityGraph = new List<Vertex>();
33     List<Vertex> allPolygonVertices = this.allPolygonVertices.Where(x
34         => !x.vector.Approx(origin, epsilon)).ToList();
35
36     endpoints = GetEndpoints(origin).ToList();
37     Dictionary<Vertex, double> angles =
38         allPolygonVertices
39         .Concat(endpoints.Select(x => x.Item1))
40         .ToDictionary(x => x, x => x.vector.Angle(origin));
41
42     // Edges are stored as the vertex with the lower index of the two
43     // defining vertices
44     IComparer<Vertex> comparer = Comparer<Vertex>.Create((a, b) =>
45     {
46         if (ReferenceEquals(a, b) || a == b) return 0;
47
48         // Based on https://github.com/trylock/visibility/blob/master
49         // /visibility/visibility.hpp Lines 17-89
50
51         Vector a1 = a.vector;
52         Vector a2 = a.Next.vector;
53         Vector b1 = b.vector;
54         Vector b2 = b.Next.vector;
```

```
49 // If there are common endpoints, let them be a1 and b1
50 if (a2.Approx(b1, epsilon) || a2.Approx(b2, epsilon)) (a1, a2
    ) = (a2, a1);
51 if (a1.Approx(b2, epsilon)) (b1, b2) = (b2, b1);
52
53 if (a1.Approx(b1, epsilon)) // If there are common endpoints
    a1 and b1 this is true
54 {
55     if (a2.Approx(b2, epsilon)) return 0; // Same Lines
56     // a and b are on opposing sides of ray from origin to
    shared point (current ray in sweep-line algorithm)
57 if (Vector.OrientationApprox(origin, a1, b2, epsilon) !=
    Vector.OrientationApprox(origin, a1, a2, epsilon))
58 {
59     throw new Exception("Attempted Change to early");
60 }
61
62 // b2 is on the same side of a as origin => b is below a
63 return Vector.OrientationApprox(a1, a2, b2, epsilon) ==
    Vector.OrientationApprox(a1, a2, origin, epsilon) ? 1
    : -1;
64 }
65 else
66 {
67     var ba1 = Vector.OrientationApprox(b1, b2, a1, epsilon);
68     var ba2 = Vector.OrientationApprox(b1, b2, a2, epsilon);
69
70     // Line Segments are on a shared line but don't have
    common endpoints
71 if (ba2 == Vector.VectorOrder.Collinear && ba1 == Vector.
    VectorOrder.Collinear)
72 {
73     // Since the line segments are on a shared line, only
    one point needs to be compared
74     return origin.DistanceSquared(a1).CompareTo(origin.
    DistanceSquared(b1));
75 }
76 else if (ba1 == ba2 // a1 and a2 are entirely above or
    below b
77         || ba1 == Vector.VectorOrder.Collinear || ba2 ==
    Vector.VectorOrder.Collinear) // or a has one
    point on b => a is entirely above or below b
78 {
79     var bOrigin = Vector.OrientationApprox(b1, b2, origin
    , epsilon);
80     return bOrigin == ba1 // a1 is on the same side of b
    as origin => a is closer
81         || bOrigin == ba2 // a2 is on the same side of b
    as origin => a is closer // Check both as one
    might be collinear
82         ? -1 : 1;
83 }
84 else // a1 and a2 are on opposing sides of b (a crosses
    the infinite line containing b) => b is entirely above
    or below a
85 {
86     return Vector.OrientationApprox(a1, a2, origin,
    epsilon) == Vector.OrientationApprox(a1, a2, b1,
```

```

        epsilon) // b1 is on the same side of a as origin
        => b is below a
        ? 1 : -1;
87
88     }
89 }
90 });
91 SortedSet<Vertex> intersections = new SortedSet<Vertex>(comparer)
92 ;
93 foreach (Vertex polygonVertex in allPolygonVertices)
94 {
95     if ((polygonVertex.Next.vector - origin).y * (polygonVertex.
96         vector - origin).y < -epsilon
97         && CalculateDistanceAtAngle(polygonVertex, origin, 0) >=
98             epsilon)
99     {
100         intersections.Add(polygonVertex);
101     }
102 }
103 List<(double min, double max)> leftTouching = new List<(double,
104     double)>();
105 List<(double min, double max)> rightTouching = new List<(double,
106     double)>();
107
108 List<(double min, double max)> GetLeft() => leftTouching;
109 List<(double min, double max)> GetRight() => rightTouching;
110
111 bool IsVisible(Vertex target)
112 {
113     if (!(target.polygon is null))
114     {
115         if (!target.BetweenNeighbors(origin)) return false;
116     }
117     if (!(originVertex.polygon is null))
118     {
119         if (!originVertex.BetweenNeighbors(target.vector)) return
120             false;
121         if (originVertex.IsNeighbor(target)) return true; //
122             Neighbours are not always visible in a reduced graph
123     }
124
125     if (intersections.Count != 0 &&
126         intersections.First().Let(x => Vector.IntersectingLines(
127             origin, target.vector, x.vector, x.Next.vector)))
128         return false;
129
130     var furthestDistance =
131         GetLeft()
132         .SelectMany(x => GetRight()
133             .Where(y =>
134                 (x.min <= y.min && y.min <= x.max)
135                 || (x.min <= y.max && y.max <= x.max)
136             ) // Only take intersections
137             .Select(y => Math.Max(x.min, y.min))
138         )
139         .Let(blocked => blocked.Any() ? blocked.Min() : double.
140             PositiveInfinity);

```



```
133
134     if (origin.Distance(target.vector) > furthestDistance) return
135         false;
136
137     return true;
138 }
139
140 (Vertex vert, double currentAngle)[] sortedAngles = angles
141     .Select(x => (x.Key, x.Value)).ToArray();
142 Array.Sort(sortedAngles, Comparer<(Vertex vert, double
143     currentAngle)>.Create((a, b) => a.currentAngle.CompareTo(b.
144     currentAngle)));
145 IEnumerable<(Vertex vert, double currentAngle)> sortedAnglesEnum
146     = sortedAngles;
147
148 // Group vertices with the same angle together
149 var vertsByAngle = new List<(List<Vertex> vertices, double
150     prevAngle, double angle, double nextAngle)>();
151 {
152     double angle;
153     double prevAngle = 0;
154     while (sortedAnglesEnum.Any())
155     {
156         angle = sortedAnglesEnum.First().currentAngle;
157         List<Vertex> buffer = sortedAnglesEnum.TakeWhile(x => x.
158             currentAngle == angle).Select(x => x.vert).ToList();
159         sortedAnglesEnum = sortedAnglesEnum.Skip(buffer.Count);
160         vertsByAngle.Add((buffer, prevAngle, angle,
161             sortedAnglesEnum.Any() ? sortedAnglesEnum.First().
162             currentAngle : Math.PI * 2));
163         prevAngle = angle;
164     }
165 }
166
167 List<Vertex> delta = new List<Vertex>();
168
169 void Add(double currentAngle, double nextAngle, Vertex first,
170     Vertex second)
171 {
172     if (first.vector.Approx(origin, epsilon) || second.vector.
173         Approx(origin, epsilon)) return; // Already handled by
174         BetweenNeighbours
175
176     // Collinear lines aren't intersections, only their position
177     // on the ray is used
178     if (Vector.OrientationApprox(origin, first.vector, second.
179         vector, epsilon) != Vector.VectorOrder.Collinear) delta.
180         Add(first);
181     leftTouching.Add(angles[first] == angles[second]
182         ? (origin.DistanceSquared(first.vector), origin.
183             DistanceSquared(second.vector)) // Squaring later
184             is cheaper than Sqrt here
185         .Let(x => x.Item1 < x.Item2 ? x : (x.Item2, x.
186             Item1))
187         : origin.DistanceSquared(first.vector).Let(x => (x, x
188             ))));
189 }
190
191 void Remove(double prevAngle, double currentAngle, Vertex first,
```

```
Vertex second)
173 {
174     if (first.vector.Approx(origin, epsilon) || second.vector.
        Approx(origin, epsilon)) return; // Already handled by
        BetweenNeighbours
175
176     // Collinear lines aren't intersections, only their position
        on the ray is used
177     if (Vector.OrientationApprox(origin, first.vector, second.
        vector, epsilon) != Vector.VectorOrder.Collinear)
        intersections.Remove(first);
178     rightTouching.Add(angles[first] == angles[second]
179         ? (origin.DistanceSquared(first.vector), origin.
            DistanceSquared(second.vector)) // Squaring later
            is cheaper than Sqrt here
180         .Let(x => x.Item1 < x.Item2 ? x : (x.Item2, x.
                Item1))
181         : origin.DistanceSquared(first.vector).Let(x => (x, x
            )))
182 }
183
184 foreach ((List<Vertex> vertices, double prevAngle, double
    currentAngle, double nextAngle) in vertsByAngle)
185 {
186     foreach (Vertex vert in vertices)
187     {
188         if (vert.polygon is null) continue;
189
190         Vertex previous = vert.Previous;
191         if (Vector.Orientation(previous.vector, vert.vector,
            origin) != Vector.VectorOrder.Clockwise) Remove(
            prevAngle, currentAngle, previous, vert);
192         else Add(currentAngle, nextAngle, previous, vert);
193
194         Vertex next = vert.Next;
195         if (Vector.Orientation(next.vector, vert.vector, origin)
            != Vector.VectorOrder.Clockwise) Remove(prevAngle,
            currentAngle, vert, next);
196         else Add(currentAngle, nextAngle, vert, next);
197     }
198
199     visibilityGraph.AddRange(vertices.Where(IsVisible));
200
201     leftTouching.Clear();
202     rightTouching.Clear();
203
204     delta.ForEach(x => intersections.Add(x));
205     delta.Clear();
206 }
207
208 var polygon = visibilityGraph.Distinct().ToList();
209 debug = debugOut;
210 return polygon;
211 }
212
213 public Dictionary<Vertex, List<Vertex>> GenerateVisibilityGraph(out
    List<(Vertex vert, double busLength)> endpoints, out List<(Vector,
    Vector)> debug)
```

```
214 {
215     var debugOut = new List<(Vector, Vector)>();
216     var endpointsOut = new List<(Vertex vert, double busLength)>();
217     var graph =
218         allPolygonVertices
219         .Concat(new[] { startingPosition })
220         .ToDictionary(x => x, x =>
221         {
222             var polygon = GenerateVisibilityPolygon(x, out var
                newEndpoints, out var newDebug);
223             debugOut.AddRange(newDebug);
224             endpointsOut.AddRange(newEndpoints);
225             return polygon;
226         });
227
228     debug = debugOut;
229     endpoints = endpointsOut;
230     return graph;
231 }
232
233 public Dictionary<Vertex, Vertex> GenerateDijkstraHeuristic(bool
    reduced, out Dictionary<Vertex, Dictionary<Vertex, double>>
    visitedNodes, out List<(Vertex vert, double busLength)> endpoints,
    out List<(Vector, Vector)> debug)
234 {
235     List<Vertex> allVertices = allPolygonVertices.Concat(new[] {
        startingPosition }).ToList();
236
237     var debugOut = new List<(Vector, Vector)>();
238     var endpointsOut = new List<(Vertex vert, double busLength)>();
239
240     Dictionary<Vertex, Func<Dictionary<Vertex, double>>> graph =
241         allVertices.ToDictionary(x => x, x =>
242         (Func<Dictionary<Vertex, double>>)(() =>
243         {
244             var polygon = GenerateVisibilityPolygon(x, out var
                newEndpoints, out var newDebug);
245             debugOut.AddRange(newDebug);
246             endpointsOut.AddRange(newEndpoints);
247             return polygon.ToDictionary(y => y, y => y.vector.
                Distance(x.vector));
248         }
249         ));
250
251     var dijkstra = Dijkstra.GenerateDijkstraHeuristicLazy(
        startingPosition, graph, endpointsOut.Select(x => x.vert).
        ToList(), out visitedNodes);
252     debug = debugOut;
253     endpoints = endpointsOut;
254     return dijkstra;
255 }
256
257 public List<Vertex> GetOptimalPath(out double characterLength, out
    double busLength, out double advantage, out List<(Vector, Vector)>
    debug)
258 {
259     var heuristic = GenerateDijkstraHeuristic(true, out var
        visitedNodes, out var endpoints, out debug);
```

```
260
261     IEnumerable<(Vertex vert, double characterLength, double
        busLength)> times = endpoints
262         .Where(x => heuristic.ContainsKey(x.vert))
263         .Select(x =>
264             (x.vert, Dijkstra.GetPathLength(startingPosition, x.vert,
                heuristic, visitedNodes), x.busLength));
265
266     Vertex min;
267     ((min, characterLength, busLength), advantage) = times.MinValue(x
        => x.characterLength / characterSpeed - x.busLength /
        busSpeed);
268     return Dijkstra.GetPath(startingPosition, min, heuristic);
269 }
```

#### Listing 7: class Dijkstra

```
1 public static Dictionary<Vertex, Vertex>
    GenerateDijkstraHeuristicLazy(Vertex start, Dictionary<Vertex,
    Func<Dictionary<Vertex, double>>> nodes, List<Vertex>
    reachingRequired, out Dictionary<Vertex, Dictionary<Vertex, double
    >> visitedNodes)
2 {
3     List<Vertex> priorityList = nodes.Keys.ToList();
4     reachingRequired = reachingRequired.Where(x => priorityList.
        Contains(x)).ToList();
5
6     Dictionary<Vertex, double> distance = new Dictionary<Vertex,
        double>();
7     Dictionary<Vertex, Vertex> path = new Dictionary<Vertex, Vertex
        >();
8     Dictionary<Vertex, Dictionary<Vertex, double>> visitedNodesOut =
        new Dictionary<Vertex, Dictionary<Vertex, double>>();
9     distance[start] = 0;
10    path[start] = start;
11
12    void Step(Vertex current)
13    {
14        foreach (var connection in (visitedNodesOut[current] = nodes[
            current]()))
15        {
16            double newDistance = connection.Value + distance[current
                ];
17            if (!distance.ContainsKey(connection.Key)) distance[
                connection.Key] = double.PositiveInfinity;
18            if (distance[connection.Key] > newDistance)
19            {
20                path[connection.Key] = current;
21                distance[connection.Key] = newDistance;
22            }
23        }
24
25        priorityList.Remove(current);
26        reachingRequired.Remove(current);
27    }
28    IComparer<Vertex> comparer = Comparer<Vertex>.Create((a, b) =>
        distance[a].CompareTo(distance[b]));
29    while (priorityList.Any()) Step(priorityList.MinValue(x =>
```

```
        distance.ContainsKey(x) ? distance[x] : double.
        PositiveInfinity).value);
30
31     visitedNodes = visitedNodesOut;
32     return path.ToDictionary(x => x.Key, x => x.Value);
33 }
34
35 public static List<Vertex> GetPath(Vertex start, Vertex end,
    Dictionary<Vertex, Vertex> heuristic)
36 {
37     List<Vertex> path = new List<Vertex>();
38     for (Vertex current = end, next = heuristic[end]; current !=
        start; current = next, next = heuristic[current]) path.Add(
        current);
39     path.Add(start);
40     return path;
41 }
42
43 public static double GetPathLength(Vertex start, Vertex end,
    Dictionary<Vertex, Vertex> heuristic, Dictionary<Vertex,
    Dictionary<Vertex, double>> visitedNodes)
44 {
45     double length = 0;
46     for (Vertex current = end, next = heuristic[end]; current !=
        start; current = next, next = heuristic[current]) length +=
        visitedNodes[next][current];
47     return length;
48 }
```