# Aufgabe 1

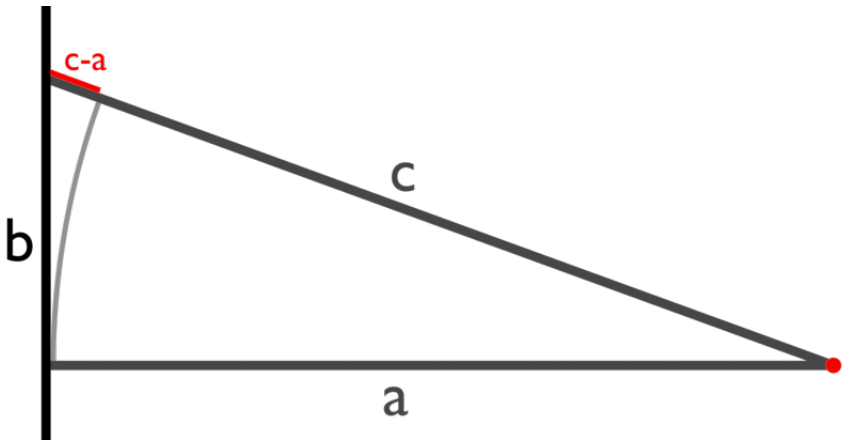
Nikolas Kilian

8. März 2019

## 1 Lösungsidee

Wenn es keine Hindernisse gibt, so ist der optimale Weg eine gerade Strecke vom Startpunkt zum Buspfad im 30° 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° Strecke offen ist, und dann diese 30° Strecke.
Um das Optimum mit Hindernissen zu finden, muss man also alle Eckpunkte bestimmen, von denen aus diese 30° 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) veränderet 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.

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

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

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

# 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))
           return false;
13     if (Origin and p are neighbours) return true;
14
15     if (Intersections is not empty and its leftmost element
           intersects the line from Origin to Target) return false;
16 }
```

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 konvex, so ist das Ergebnis immer false. Wenn der Rückgabewert dieser Methode false ist, so sind in einem reduziertem Sichtbarkeitsgraph die beiden Punkte nicht verbunden.

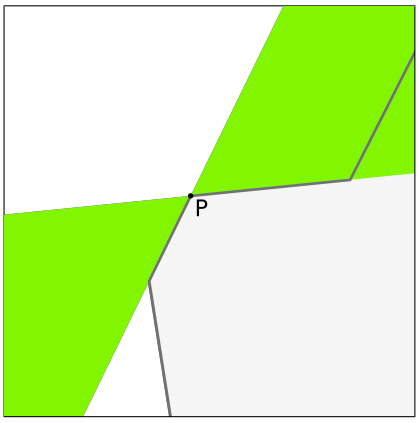



Abbildung 1: BetweenNeighbours

Um das durchgehen unendlich dünner Wege zu verhindern, speichere ich die hinzugefügten/entfernten Kanten, errechne die Strecke die sie auf der Strecke zum aktuellem 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 genreriere 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 Priotitätsliste 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 Startezeit. Dieser Weg ist der optimale Weg, und somit das Ergebnis.

# 3 Beispiele

## 3.1 Beispiel 1

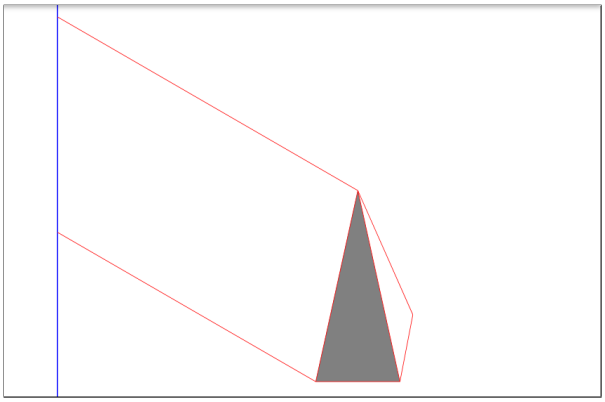


Abbildung 2: Sichtbarkeitsgraph

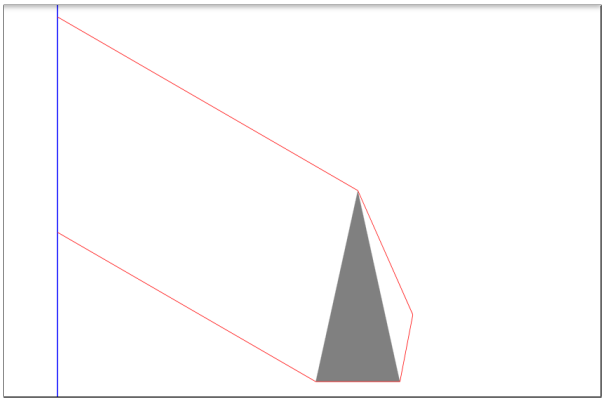


Abbildung 3: Dijkstra Heuristik

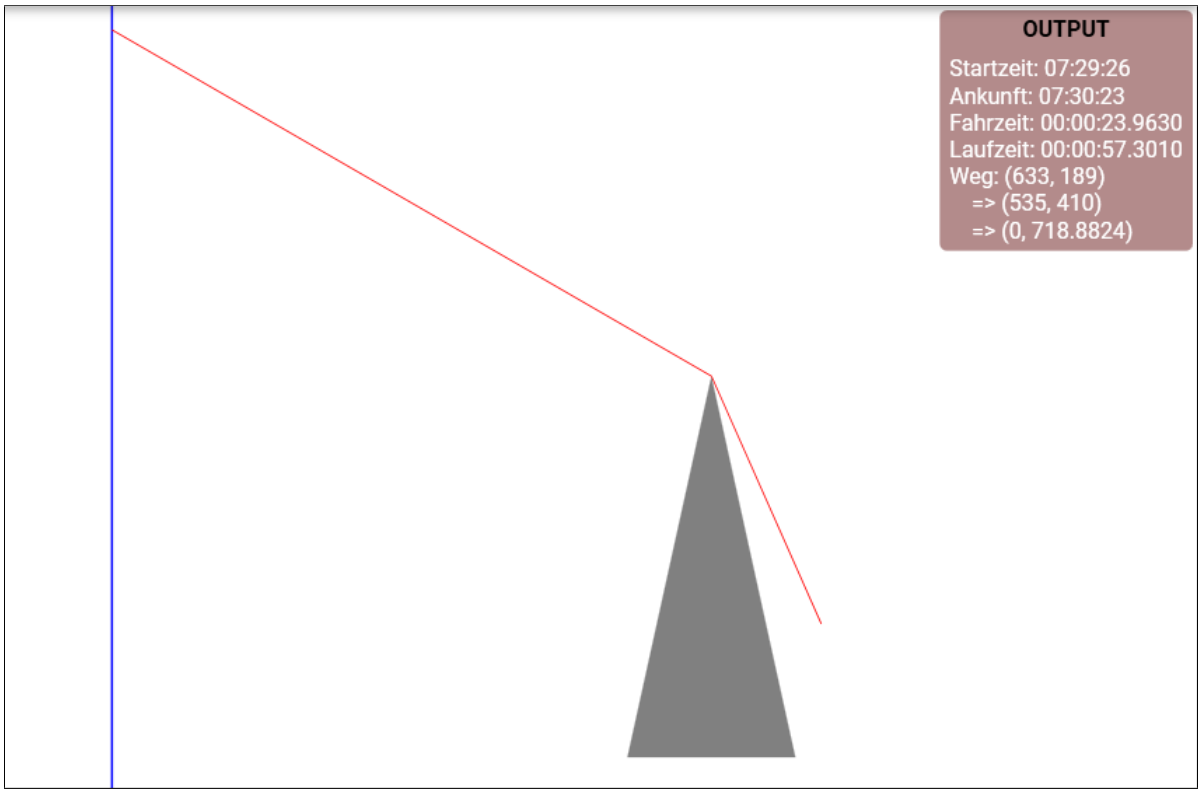



Abbildung 4: optimaler Weg

## 3.2 Beispiel 2

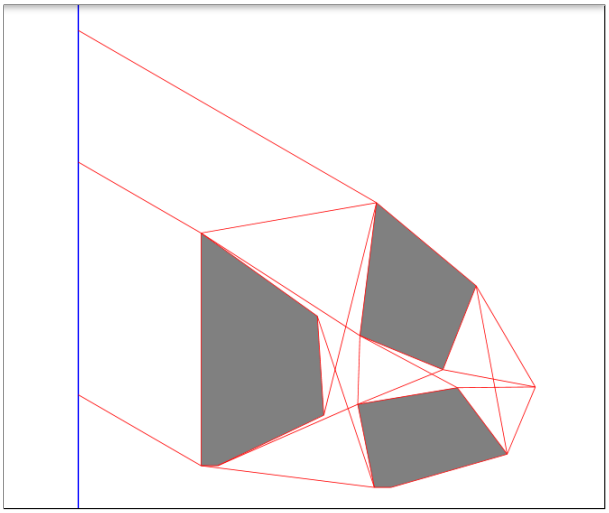


Abbildung 5: Sichtbarkeitsgraph

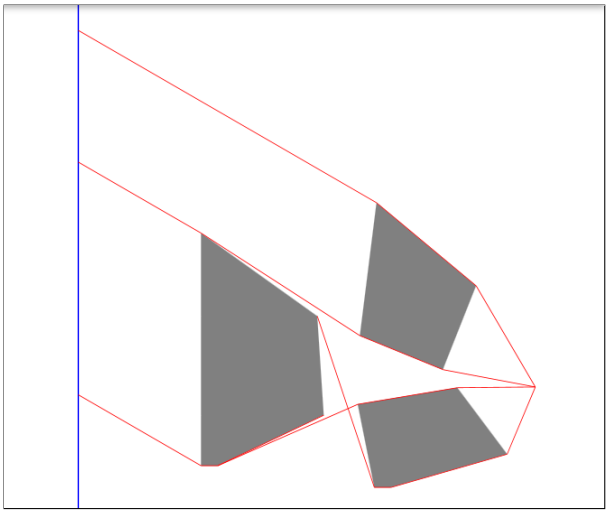


Abbildung 6: Dijkstra Heuristik

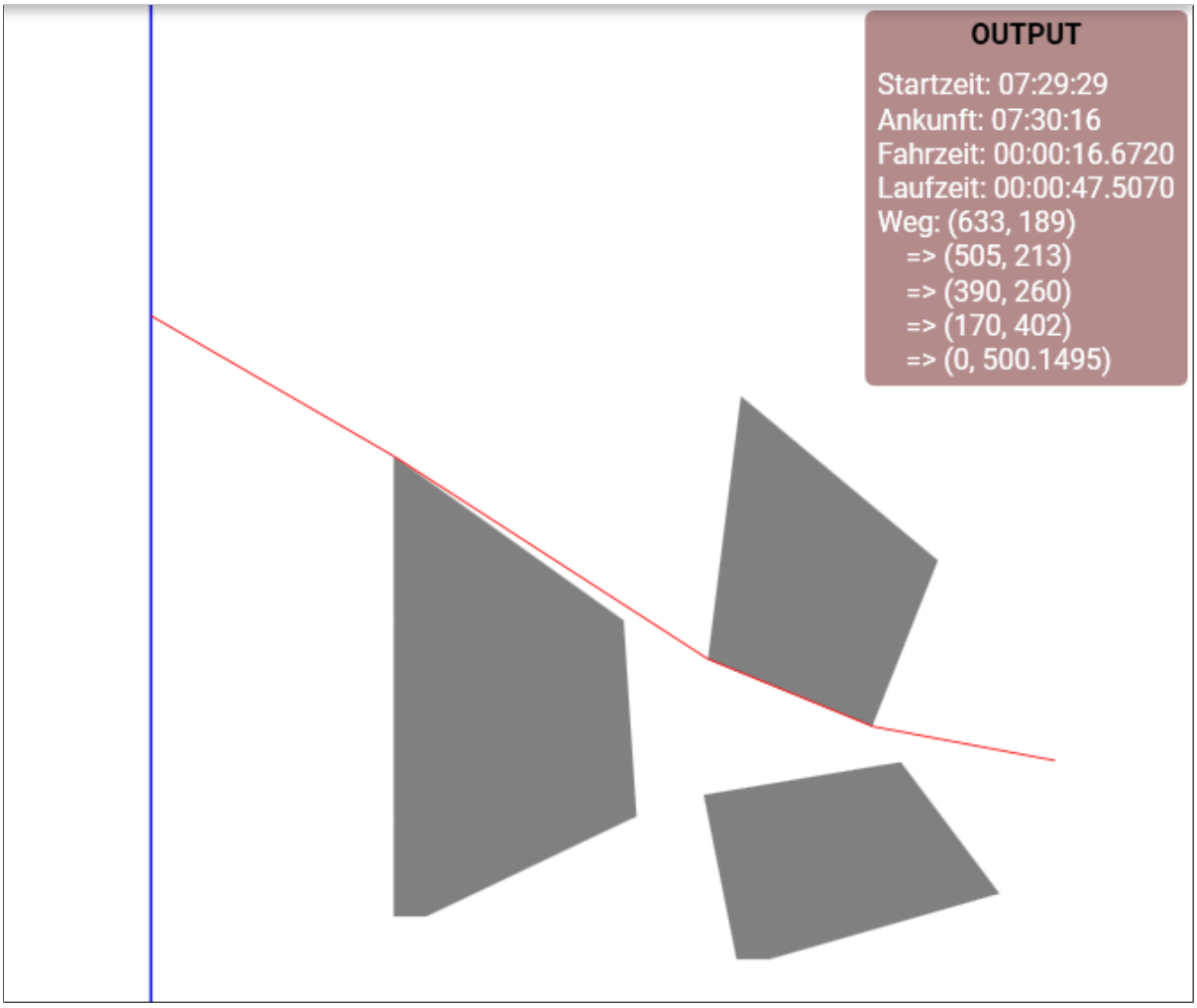



Abbildung 7: optimaler Weg

## 3.3 Beispiel 3

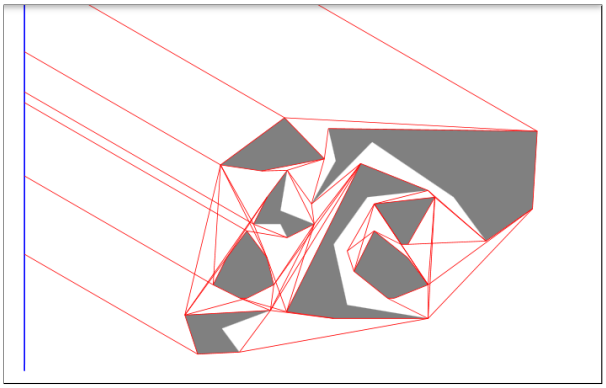


Abbildung 8: Sichtbarkeitsgraph

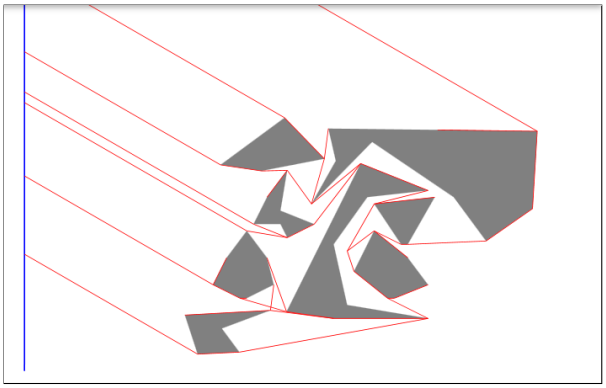


Abbildung 9: Dijkstra Heuristik

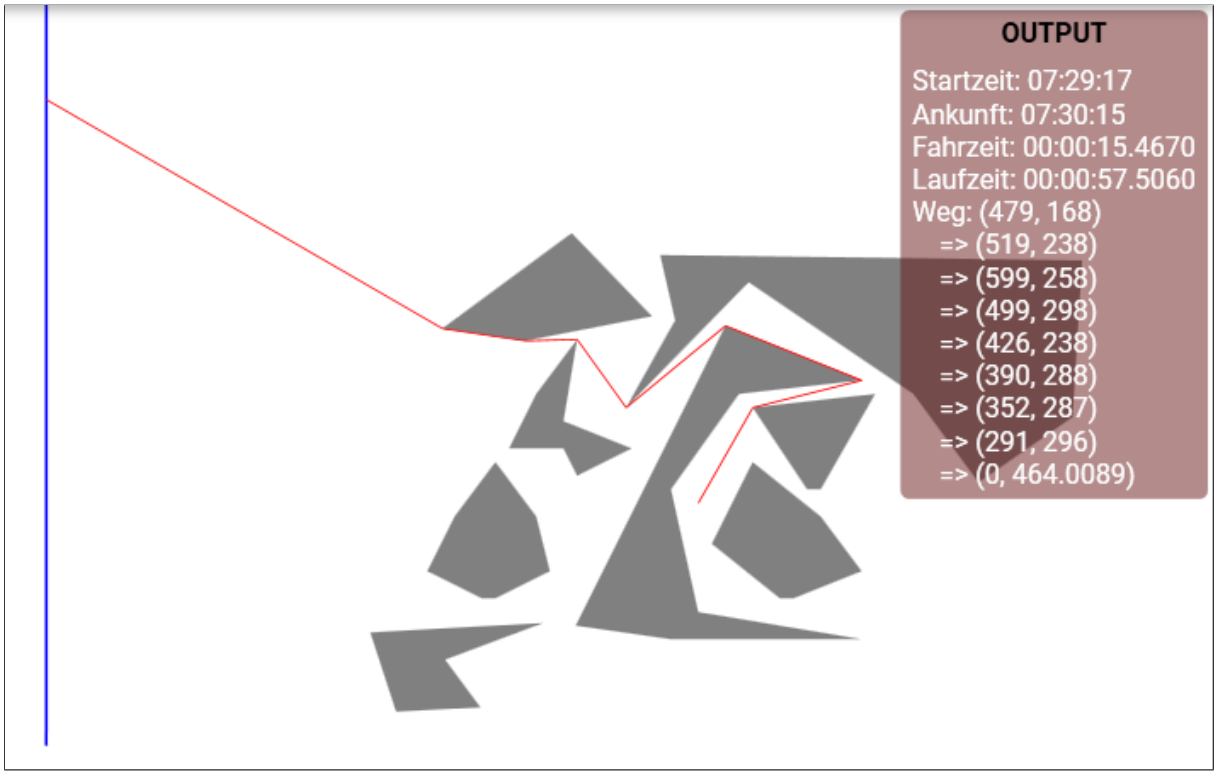



Abbildung 10: optimaler Weg

## 3.4 Beispiel 4

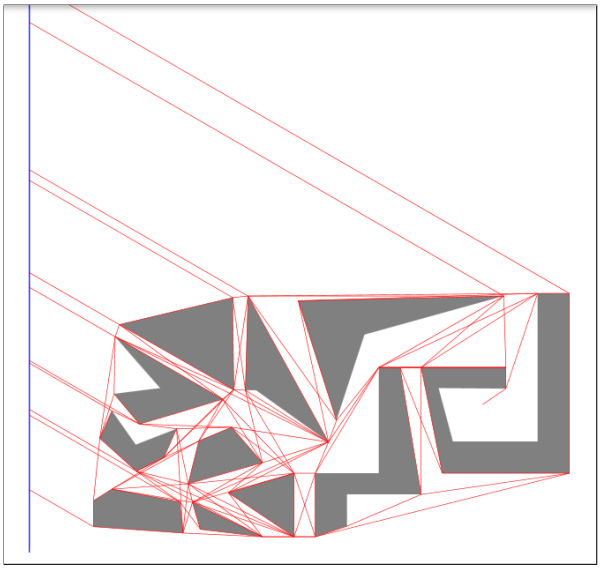


Abbildung 11: Sichtbarkeitsgraph

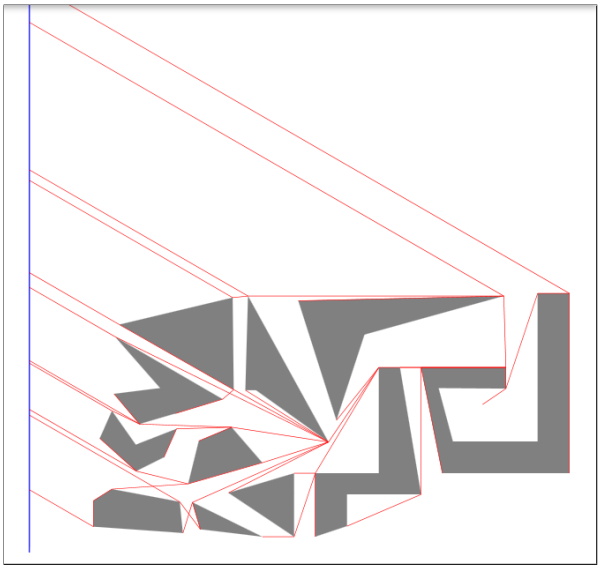


Abbildung 12: Dijkstra Heuristik

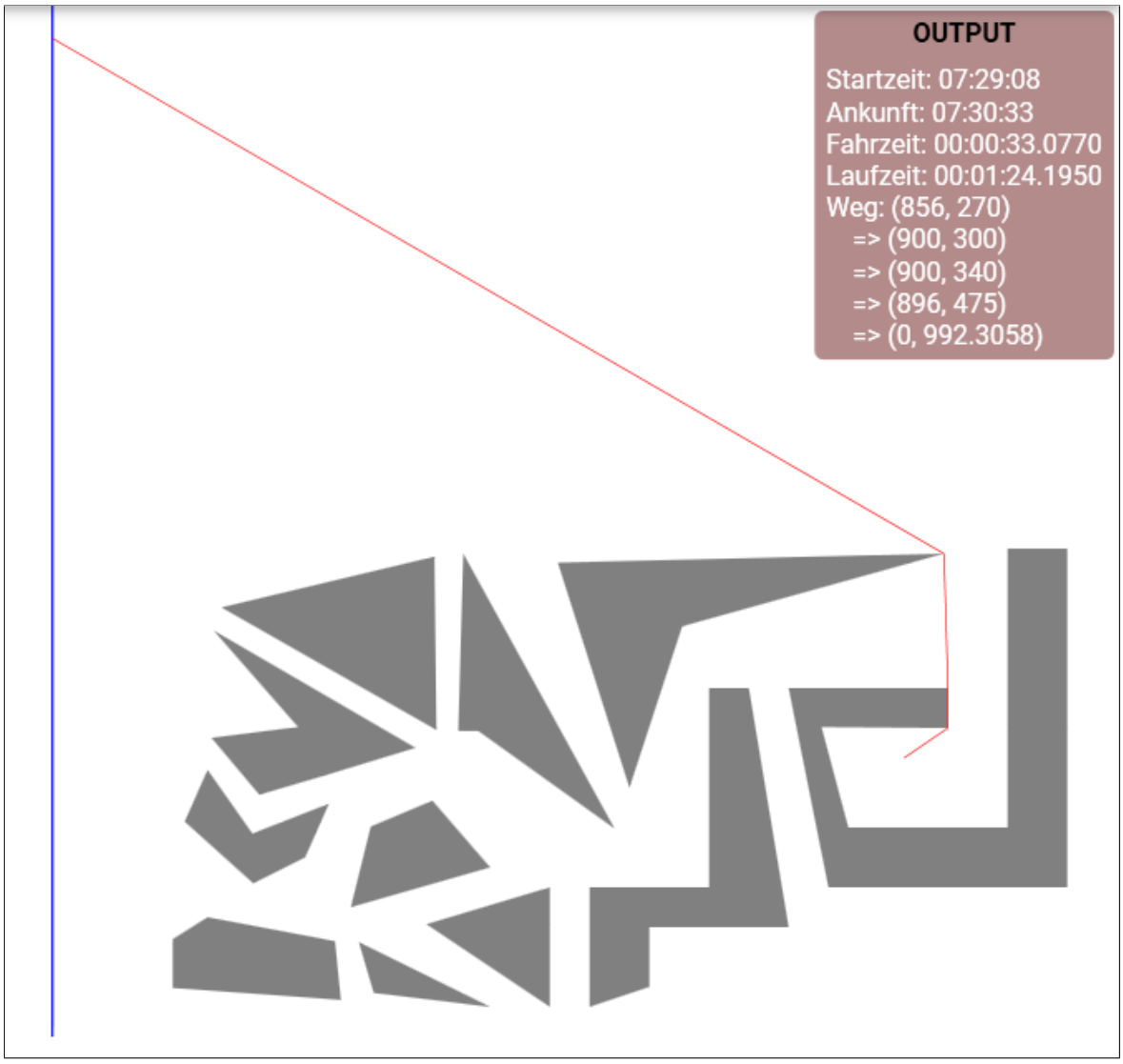



Abbildung 13: optimaler Weg

## 3.5 Beispiel 5

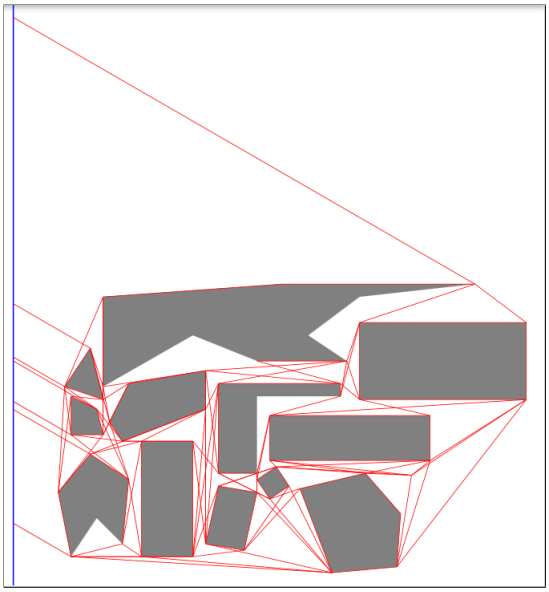


Abbildung 14: Sichtbarkeitsgraph

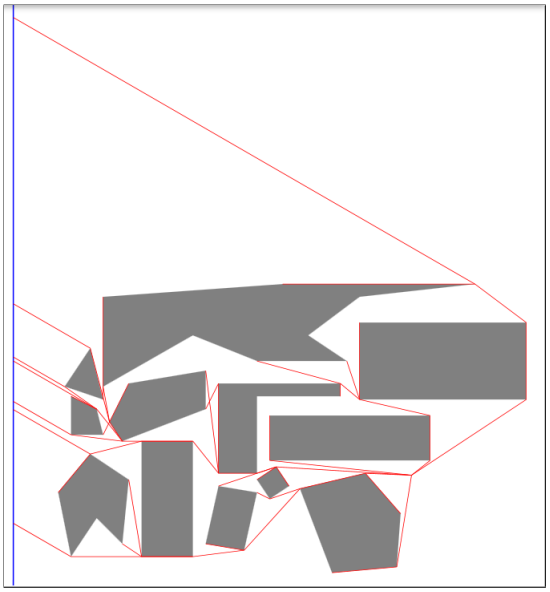


Abbildung 15: Dijkstra Heuristik

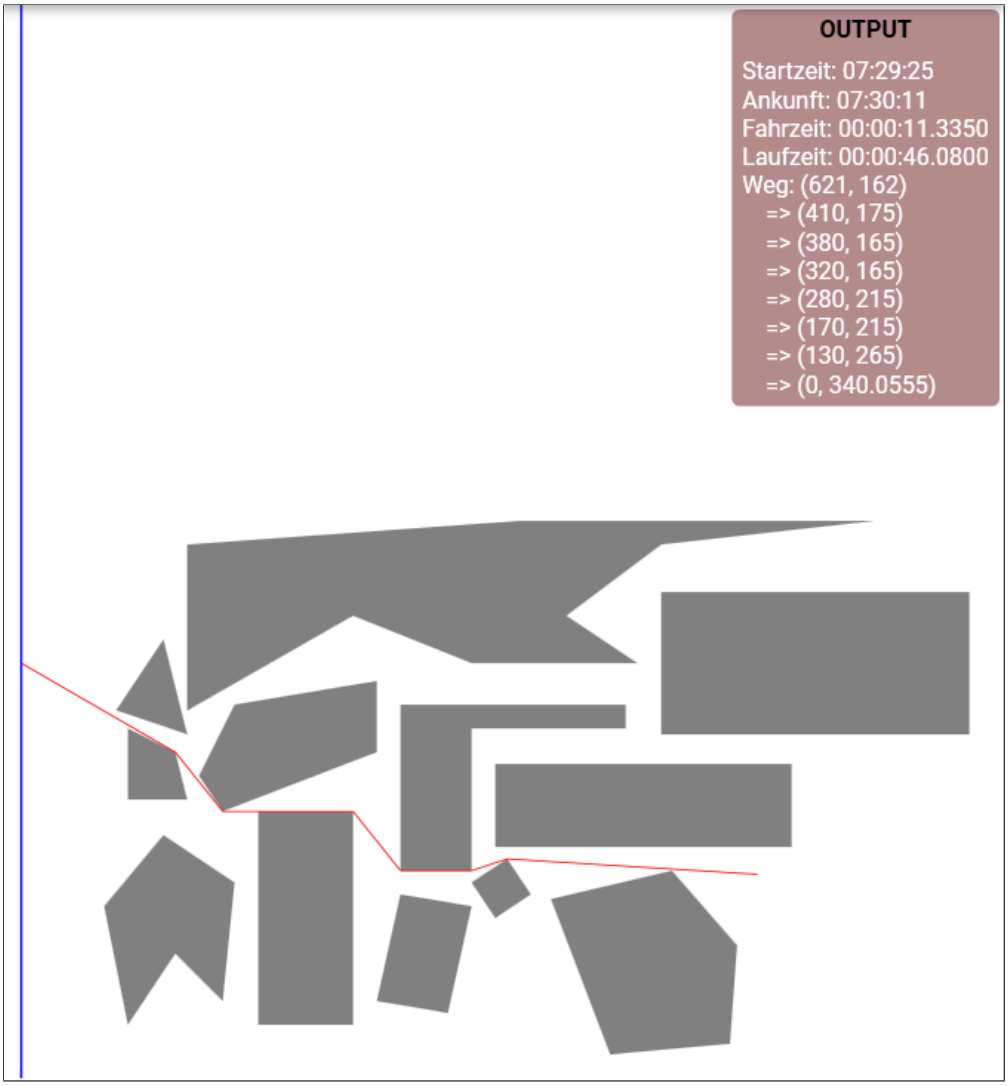



Abbildung 16: optimaler Weg

# 4 Code

**Listing 1: Results**

```
DateTime start = DateTime.Now.Let(x => new DateTime(x.Year, x.Month,
    x.Day, 7, 30, 0));

output.Text =
@$"Startzeit: {(start - TimeSpan.FromSeconds(advantage))}
Ankunft: {(start + TimeSpan.FromSeconds(busLength / map.busSpeed))}
Fahrzeit: {TimeSpan.FromSeconds(busLength / map.busSpeed)}
Laufzeit: {TimeSpan.FromSeconds(characterLength / map.characterSpeed)
    }
Weg: {string.Join("=>", Enumerable.Reverse(optimalPath).Select(x => $
    "({x.vector.x}, {x.vector.y})"))}";
```

**Listing 2: class Map**

```
\begin{Csharp}
public Polygon[] polygons;
public Vector[] busPath;
public Vertex startingPosition;
public List<Vertex> allPolygonVertices;
public double busSpeed, characterSpeed, busApproachConstant;

public void SetSpeed(double characterSpeed, double busSpeed)
{
    this.characterSpeed = characterSpeed;
    this.busSpeed = busSpeed;
    busApproachConstant = characterSpeed / Math.Sqrt(busSpeed *
        busSpeed - characterSpeed * characterSpeed);
}

public IEnumerable<Vector> GetEndpoints(Vector dot)
// Returns all endpoints of the direct paths (30 degree angle) from
    the given Vector to the bus path

public double GetBusLength(Vector vec)
// Calculates the distance the bus drives to reach a given point on
    its track

public double CalculateDistanceAtAngle(Vertex vertex, Vector origin,
    double angle)
// Calculates the distance from origin to the intersection between a
    ray from origin with a given angle, and the line containing vertex
     and vertex.Next

public double epsilon = 1E-15;

public List<Vertex> GenerateVisibilityPolygon(Vertex originVertex,
    out List<Vertex> endpoints, out List<(Vector, Vector)> debug)
{
    Vector origin = originVertex.vector;

    List<Vertex> visibilityGraph = new List<Vertex>();
    List<Vertex> allPolygonVertices = this.allPolygonVertices.Where(x
        => !x.vector.Approx(origin, epsilon)).ToList();
```

```
32
33      endpoints = GetEndpoints(origin).Select(x => new Vertex(x)).
            ToList();
34      Dictionary<Vertex, double> angles =
35          allPolygonVertices
36          .Concat(endpoints)
37          .ToDictionary(x => x, x => x.vector.Angle(origin));
38
39      // Edges are stored as the vertex with the lower index of the two
            defining vertices
40      IComparer<Vertex> comparer = Comparer<Vertex>.Create((a, b) =>
41      {
42          if (ReferenceEquals(a, b) || a == b) return 0;
43
44          // Based on https://github.com/trylock/visibility/blob/master
                /visibility/visibility.hpp Lines 17-89
45
46          Vector a1 = a.vector;
47          Vector a2 = a.Next.vector;
48          Vector b1 = b.vector;
49          Vector b2 = b.Next.vector;
50
51          // If there are common endpoints, let them be a1 and b1
52          if (a2.Approx(b1, epsilon) || a2.Approx(b2, epsilon)) (a1, a2
                ) = (a2, a1);
53          if (a1.Approx(b2, epsilon)) (b1, b2) = (b2, b1);
54
55          if (a1.Approx(b1, epsilon)) // If there are common endpoints
                a1 and b1 this is true
56          {
57              if (a2.Approx(b2, epsilon)) return 0; // Same Lines
58              // a and b are on opposing sides of ray from origin to
                    shared point (current ray in sweep-line algorithm)
59              if (Vector.OrientationApprox(origin, a1, b2, epsilon) !=
                    Vector.OrientationApprox(origin, a1, a2, epsilon))
60              {
61                  throw new Exception("Attempted Change to early");
62              }
63
64              // b2 is on the same side of a as origin => b is below a
65              return Vector.OrientationApprox(a1, a2, b2, epsilon) ==
                    Vector.OrientationApprox(a1, a2, origin, epsilon) ? 1
                    : -1;
66          }
67          else
68          {
69              var ba1 = Vector.OrientationApprox(b1, b2, a1, epsilon);
70              var ba2 = Vector.OrientationApprox(b1, b2, a2, epsilon);
71
72              // Line Segments are on a shared line but don't have
                    common endpoints
73              if (ba2 == Vector.VectorOrder.Collinear && ba1 == Vector.
                    VectorOrder.Collinear)
74              {
75                  // Since the line segments are on a shared line, only
                        one point needs to be compared
76                  return origin.DistanceSquared(a1).CompareTo(origin.
                        DistanceSquared(b1));
```

```
77              }
78              else if (ba1 == ba2 // a1 and a2 are entirely above or
                    below b
79                   || ba1 == Vector.VectorOrder.Collinear || ba2 ==
                        Vector.VectorOrder.Collinear) // or a has one
                        point on b => a is entirely above or below b
80              {
81                  var bOrigin = Vector.OrientationApprox(b1, b2, origin
                        , epsilon);
82                  return bOrigin == ba1 // a1 is on the same side of b
                        as origin => a is closer
83                      || bOrigin == ba2 // a2 is on the same side of b
                        as origin => a is closer // Check both as one
                        might be collinear
84                      ? -1 : 1;
85              }
86              else // a1 and a2 are on opposing sides of b (a crosses
                    the infinite line containing b) => b is entirely above
                     or below a
87              {
88                  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
89                      ? 1 : -1;
90              }
91          }
92      });
93      SortedSet<Vertex> intersections = new SortedSet<Vertex>(comparer)
            ;
94
95      foreach (Vertex polygonVertex in allPolygonVertices)
96      {
97          if ((polygonVertex.Next.vector - origin).y * (polygonVertex.
                vector - origin).y < -epsilon
98              && CalculateDistanceAtAngle(polygonVertex, origin, 0) >=
                    epsilon)
99          {
100             intersections.Add(polygonVertex);
101         }
102     }
103
104     List<(double min, double max)> leftTouching = new List<(double,
            double)>();
105     List<(double min, double max)> rightTouching = new List<(double,
            double)>();
106
107     List<(double min, double max)> GetLeft() => leftTouching;
108     List<(double min, double max)> GetRight() => rightTouching;
109
110     bool IsVisible(Vertex target)
111     {
112         if (!(target.polygon is null))
113         {
114             if (!target.BetweenNeighbors(origin)) return false;
115         }
116         if (!(originVertex.polygon is null))
117         {
```

```
118             if (!originVertex.BetweenNeighbors(target.vector)) return
                    false;
119             if (originVertex.IsNeighbor(target)) return true; //
                    Neighbours are not always visible in a reduced graph
120         }
121
122         if (intersections.Count != 0 &&
123             intersections.First().Let(x => Vector.IntersectingLines(
                    origin, target.vector, x.vector, x.Next.vector)))
                    return false;
124
125         var furthestDistance =
126             GetLeft()
127             .SelectMany(x => GetRight()
128                 .Where(y =>
129                     (x.min <= y.min && y.min <= x.max)
130                     || (x.min <= y.max && y.max <= x.max)
131                 ) // Only take intersections
132                 .Select(y => Math.Max(x.min, y.min))
133             )
134             .Let(blocked => blocked.Any() ? blocked.Min() : double.
                    PositiveInfinity);
135
136         if (origin.Distance(target.vector) > furthestDistance) return
                false;
137
138         return true;
139     }
140
141     (Vertex vert, double currentAngle)[] sortedAngles = angles
142         .Select(x => (x.Key, x.Value)).ToArray();
143     Array.Sort(sortedAngles, Comparer<(Vertex vert, double
        currentAngle)>.Create((a, b) => a.currentAngle.CompareTo(b.
        currentAngle)));
144     IEnumerable<(Vertex vert, double currentAngle)> sortedAnglesEnum
        = sortedAngles;
145
146     // Group vertices with the same angle together
147     var vertsByAngle = new List<(List<Vertex> vertices, double
        prevAngle, double angle, double nextAngle)>();
148     {
149         double angle;
150         double prevAngle = 0;
151         while (sortedAnglesEnum.Any())
152         {
153             angle = sortedAnglesEnum.First().currentAngle;
154             List<Vertex> buffer = sortedAnglesEnum.TakeWhile(x => x.
                    currentAngle == angle).Select(x => x.vert).ToList();
155             sortedAnglesEnum = sortedAnglesEnum.Skip(buffer.Count);
156             vertsByAngle.Add((buffer, prevAngle, angle,
                    sortedAnglesEnum.Any() ? sortedAnglesEnum.First().
                    currentAngle : Math.PI * 2));
157             prevAngle = angle;
158         }
159     }
160
161     List<Vertex> delta = new List<Vertex>();
162
```

```
163     void Add(double currentAngle, double nextAngle, Vertex first,
            Vertex second)
164     {
165         if (first.vector.Approx(origin, epsilon) || second.vector.
                Approx(origin, epsilon)) return; // Already handeled by
                BetweenNeighbours
166 
167         // Collinear lines aren't intersections, only their position
                on the ray is used
168         if (Vector.OrientationApprox(origin, first.vector, second.
                vector, epsilon) != Vector.VectorOrder.Collinear) delta.
                Add(first);
169         leftTouching.Add(angles[first] == angles[second]
170                 ? (origin.DistanceSquared(first.vector), origin.
                        DistanceSquared(second.vector)) // Squaring later
                        is cheaper than Sqrt here
171                     .Let(x => x.Item1 < x.Item2 ? x : (x.Item2, x.
                        Item1))
172                 : origin.DistanceSquared(first.vector).Let(x => (x, x
                    )));
173     }
174     void Remove(double prevAngle, double currentAngle, Vertex first,
            Vertex second)
175     {
176         if (first.vector.Approx(origin, epsilon) || second.vector.
                Approx(origin, epsilon)) return; // Already handeled by
                BetweenNeighbours
177 
178         // Collinear lines aren't intersections, only their position
                on the ray is used
179         if (Vector.OrientationApprox(origin, first.vector, second.
                vector, epsilon) != Vector.VectorOrder.Collinear)
                intersections.Remove(first);
180         rightTouching.Add(angles[first] == angles[second]
181                 ? (origin.DistanceSquared(first.vector), origin.
                        DistanceSquared(second.vector)) // Squaring later
                        is cheaper than Sqrt here
182                     .Let(x => x.Item1 < x.Item2 ? x : (x.Item2, x.
                        Item1))
183                 : origin.DistanceSquared(first.vector).Let(x => (x, x
                    )));
184     }
185 
186     foreach ((List<Vertex> vertices, double prevAngle, double
            currentAngle, double nextAngle) in vertsByAngle)
187     {
188         foreach (Vertex vert in vertices)
189         {
190             if (vert.polygon is null) continue;
191 
192             Vertex previous = vert.Previous;
193             if (Vector.Orientation(previous.vector, vert.vector,
                    origin) != Vector.VectorOrder.Clockwise) Remove(
                    prevAngle, currentAngle, previous, vert);
194             else Add(currentAngle, nextAngle, previous, vert);
195 
196             Vertex next = vert.Next;
197             if (Vector.Orientation(next.vector, vert.vector, origin)
```

```
                != Vector.VectorOrder.Clockwise) Remove(prevAngle,
                currentAngle, vert, next);
198             else Add(currentAngle, nextAngle, vert, next);
199         }
200
201         visibilityGraph.AddRange(vertices.Where(IsVisible));
202
203         leftTouching.Clear();
204         rightTouching.Clear();
205
206         delta.ForEach(x => intersections.Add(x));
207         delta.Clear();
208     }
209
210     var polygon = visibilityGraph.Distinct().ToList();
211     return polygon;
212 }
213
214 public Dictionary<Vertex, Vertex> GenerateDijkstraHeuristic(bool
    reduced, out Dictionary<Vertex, Dictionary<Vertex, double>>
    visitedNodes, out List<Vertex> endpoints)
215 {
216     List<Vertex> allVertices = allPolygonVertices.Concat(new[] {
        startingPosition }).ToList();
217
218     var endpointsOut = new List<Vertex>();
219
220     Dictionary<Vertex, Func<Dictionary<Vertex, double>>> graph =
221         allVertices.ToDictionary(x => x, x =>
222         (Func<Dictionary<Vertex, double>>)(() =>
223             {
224                 var polygon = GenerateVisibilityPolygon(x, out var
                    newEndpoints);
225                 endpointsOut.AddRange(newEndpoints);
226                 return polygon.ToDictionary(y => y, y => y.vector.
                    Distance(x.vector));
227             }
228         ));
229
230     var dijkstra = Dijkstra.GenerateDijkstraHeuristicLazy(
        startingPosition, graph, endpointsOut, out visitedNodes);
231     endpoints = endpointsOut;
232     return dijkstra;
233 }
234
235 public List<Vertex> GetOptimalPath(out double characterLength, out
    double busLength, out double advantage, out List<(Vector, Vector)>
     debug)
236 {
237     var heuristic = GenerateDijkstraHeuristic(true, out var
        visitedNodes, out var endpoints, out debug);
238
239     IEnumerable<(Vertex vert, double characterLength, double
        busLength)> times = endpoints
240         .Where(x => heuristic.ContainsKey(x))
241         .Select(x =>
242             (x, Dijkstra.GetPathLength(startingPosition, x, heuristic
                , visitedNodes), GetBusLength(x.vector)));
```

```
243
244     Vertex min;
245     ((min, characterLength, busLength), advantage) = times.MinValue(x
            => x.characterLength / characterSpeed - x.busLength /
            busSpeed);
246     return Dijkstra.GetPath(startingPosition, min, heuristic);
247 }
```

**Listing 3: 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 }
```