

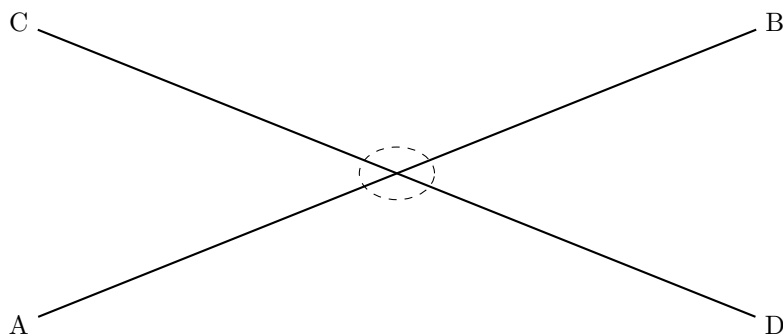
1 Lösungsidee

Die Lösungsidee besteht darin, den Dijkstra Algorithmus derart umzugestalten, dass er den Weg mit der geringsten Zahl an Abbiegungen statt der geringsten Länge sucht. Zuerst wird der kürzeste Weg berechnet, woraufhin dessen Länge und die Anzahl der darin enthaltenen Abbiegungen bestimmt wird. Daraufhin beginnt die Ermittlung des endgültigen Weges.

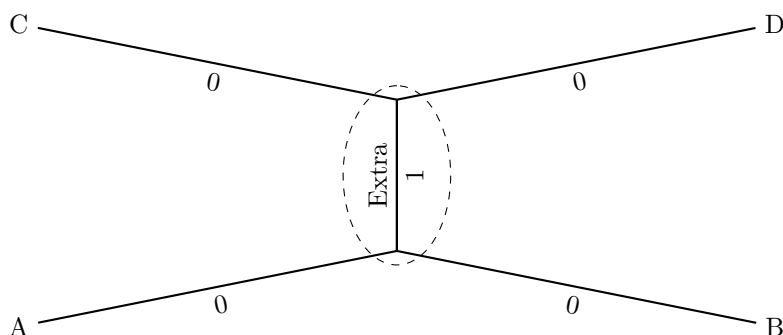
Grundlegend verläuft dieser Prozess wie der normale Dijkstra Algorithmus, jedoch mit ein paar kleinen Unterschieden: Der erste Unterschied besteht darin, dass der Algorithmus alle Wege mit einer Länge größer der Länge des kürzesten Wegs plus die erlaubte Extrastrecke (also +15% oder +30%) eliminiert. Der Hauptunterschied besteht jedoch darin, dass Dijkstra hierbei nicht nach Länge, sondern nach Anzahl an Abbiegungen optimiert.

1.1 Modifizierter Dijkstra im Detail

Um Dijkstra im Hinblick auf die minimale Anzahl an Abbiegungen umzudefinieren, könnte die zugrunde liegende Kostenfunktion anstelle der Länge des Graphen die Existenz von Abbiegungen betrachten, entweder mit einer '1' für eine Abbiegung oder einer '0' für eine Gerade. Dabei fällt auf, dass zur Berechnung der Kosten für eine Kante auch die im Weg vorangehende Kante betrachtet werden muss. Dies bedeutet aber, dass Dijkstra nicht in seiner ursprünglichen Form verwendet werden kann. Es könnte ansonsten der Fall auftreten, dass Knoten des Graphen von einem anderen Weg aus erneut besucht würden, was die Kosten der Kanten verändern würde, die mit dem Knoten verbunden sind.



Um dieses Problem zu umgehen, wird der Graph verändert: Zwar werden Kreuzungen innerhalb des Graphen immer noch als Knoten dargestellt, jedoch gibt es für jede Kreuzung mehrere Knoten, nämlich jeweils einen für jede geradlinige Straße, die in der Kreuzung vorhanden ist. Alle Knoten einer Kreuzung sind dabei verbunden durch Kanten mit Kosten von 1.



Die Priority-Queue des Dijkstra-Algorithmus ist hierbei primär nach Anzahl an Abbiegungen und sekundär nach Länge sortiert.

Der endgültige Algorithmus geht dann wie folgt vor:

1. Initialisiere eine leere Priority-Queue von Wegen.

2. Füge den Startpunkt als Weg ohne Länge und Abbiegungen hinzu.
3. Nimm den obersten Weg aus der Priority-Queue (mit den wenigsten Abbiegungen und kürzester Länge).
4. Ermittle alle Wege, die aus diesem entspringen und noch nicht vorher betrachtet wurden.
5. Speichere alle Wege zu einer Kreuzung (inklusive Richtung, in der diese befahren wird), die besser als die bisherigen Wege sind und füge diese der Priority-Queue hinzu. Dabei werden eingefügte Elemente primär nach Abbiegung und sekundär nach Weglänge sortiert werden.
6. Wiederhole 3-5, bis das Ziel an der Spitze der Priority-Queue ist.

2 Umsetzung

Der Algorithmus wurde in C# 8.0 mit .NET Core 3.1 implementiert. Es wurde als `Library OptimizedPriorityQueue`¹ verwendet. Diese implementiert eine Priority Queue, die in Dijkstra verwendet wird. Der Code ist in zwei Projekte geteilt; `Afg3Abbiegen.GUI` und `Afg3Abbiegen`.

`Afg3Abbiegen.GUI` kümmert sich um das User-Interface und ruft `Afg3Abbiegen` auf, das den eigentlichen Algorithmus enthält. `Afg3Abbiegen` definiert einige Typen:

Vector2Int stellt einen zweidimensionalen Vektor mit Integerkomponenten dar.

DirectedVector2Int stellt die Kombination einer **Vector2Int** Position und einer genormten Richtung dar.

Street definiert eine Straße zwischen zwei **Vector2Int** Endpunkten.

MapParser ist zuständig für das Einlesen der Beispieldateien.

EnumerableExtensions definiert eine Erweiterungsmethode für `IEnumerable<Vector2Int>`, die die Anzahl an Abbiegung zählt und eine, die die .

Map stellt eine Ansammlung aus Straßen mit Start und Ende dar und enthält den Hauptalgorithmus.

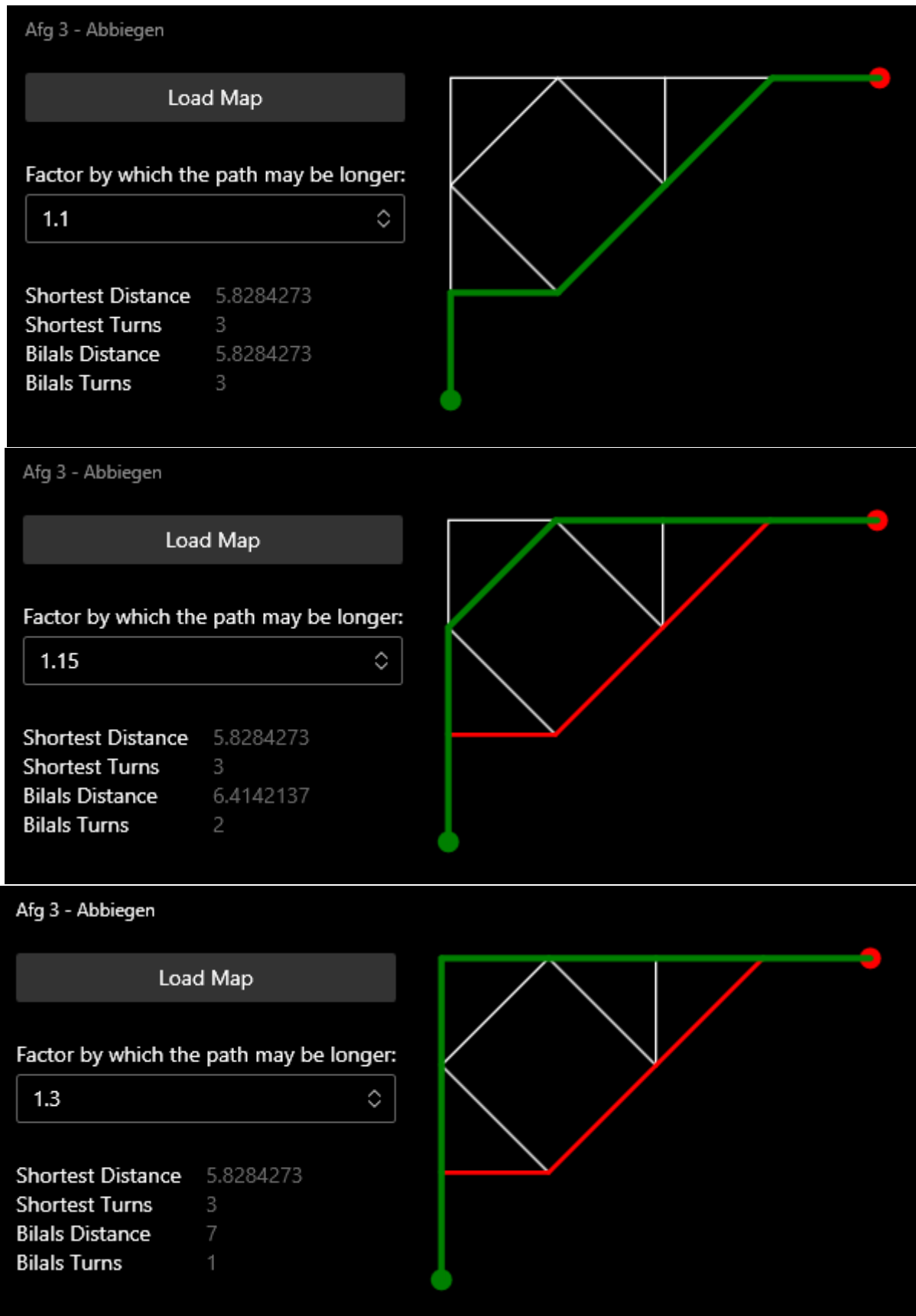
Map definiert dabei zwei Hauptmethoden **ShortestPath** und **BilalsPath**. **ShortestPath** ermittelt via Dijkstra den kürzesten Weg zwischen Start und Ende. **BilalsPath** ermittelt via dem obigen Algorithmus den in der Aufgabenstellung beschriebenen Weg.

Zur Verwendung des Programms gibt es ein GUI. Zuerst muss mit "Load Map" eine der Beispieldateien geladen werden, danach kann unter "Factor by which the path may be longer" der Faktor, um den Bilals Weg länger als der kürzeste Weg sein darf eingegeben werden, also 1.15 für +15% oder 1.30 für +30%. Der kürzeste Weg ist in rot dargestellt, Bilals Weg ist grün dargestellt. Der Startpunkt ist grün markiert, das Ziel rot.

¹<https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp>

3 Beispiele

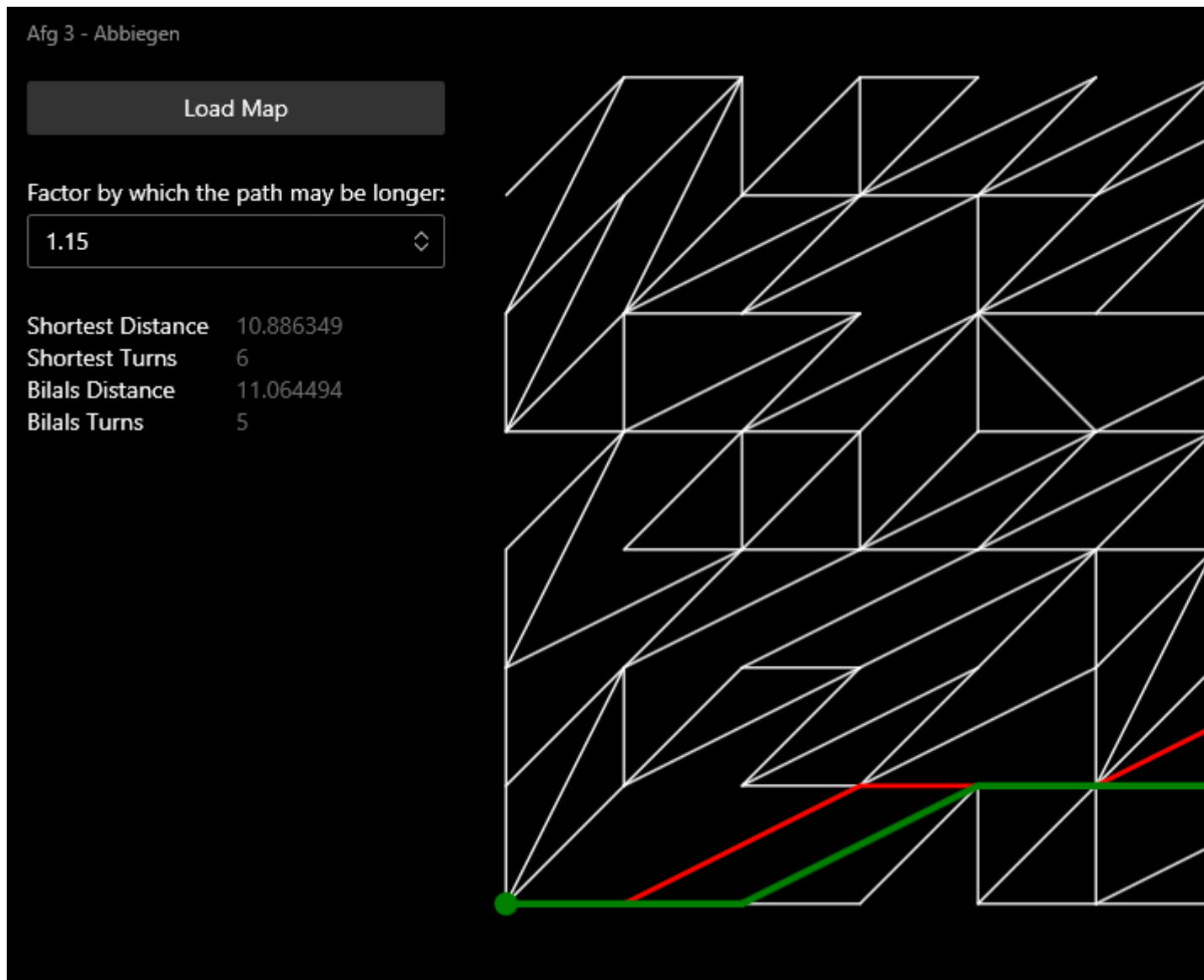
3.1

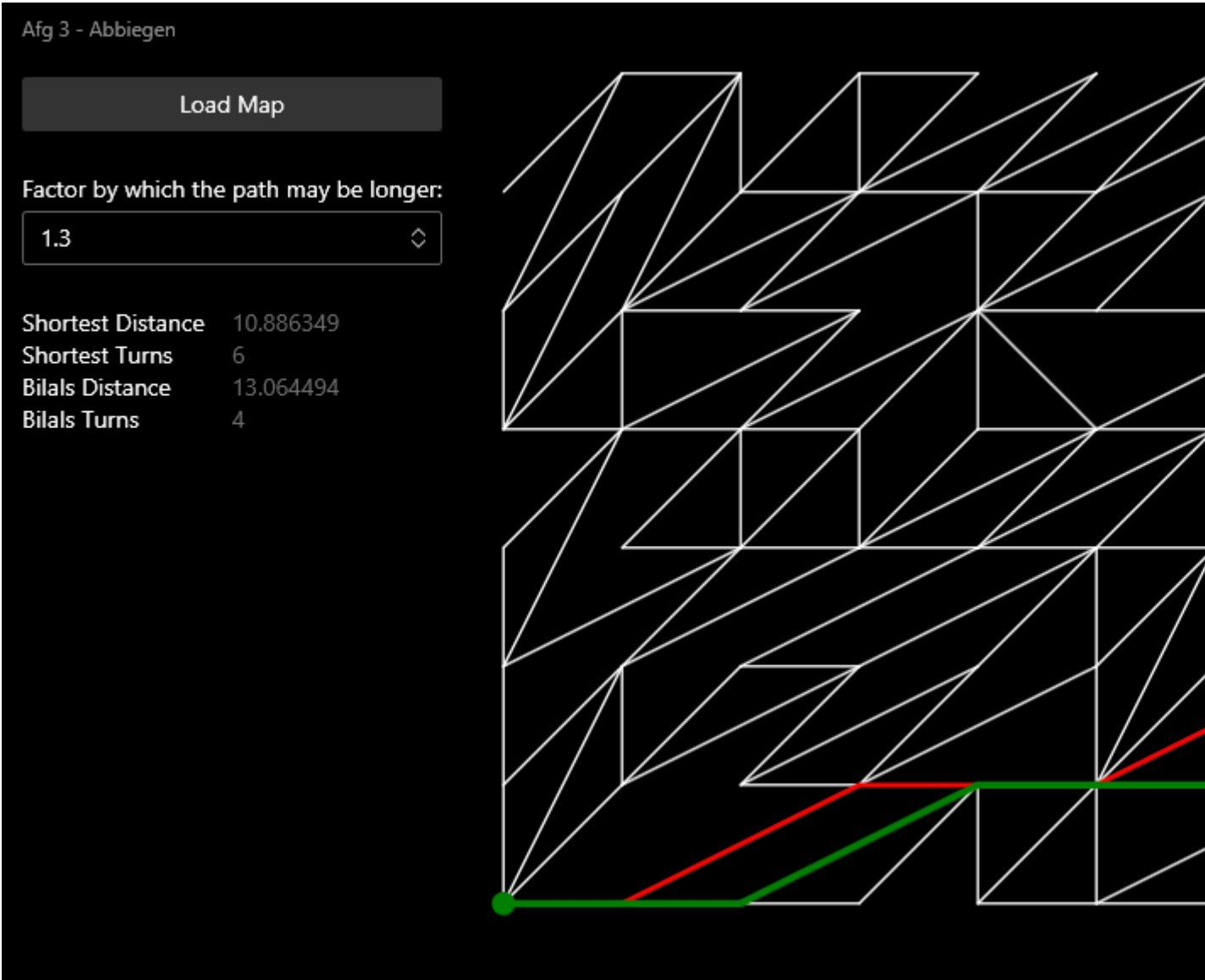


3.2

[illegible]

3.3





Afg 3 - Abbiegen

Load Map

Factor by which the path may be longer: 1.5

Shortest Distance	10.886349
Shortest Turns	6
Bilals Distance	15.944272
Bilals Turns	3

3.4

Afg 3 - Abbiegen

Load Map

Factor by which the path may be longer: 1.02

Shortest Distance 17.122417
 Shortest Turns 7
 Bilals Distance 17.300564
 Bilals Turns 6

Afg 3 - Abbiegen

Load Map

Factor by which the path may be longer:

1.04

Shortest Distance 17.122417

Shortest Turns 7

Bilals Distance 17.708202

Bilals Turns 5

Afg 3 - Abbiegen

Load Map

Factor by which the path may be longer:

1.3

Shortest Distance 17.122417

Shortest Turns 7

Bilals Distance 17.886349

Bilals Turns 4

4 Code

```

1  /// <summary>
2  /// Represents a street between two points.
3  /// </summary>
4  public readonly struct Street : IEquatable<Street>
5  {
6      /// <summary>
7      /// The start of the street.
8      /// </summary>
9      public readonly Vector2Int Start;
10
11     /// <summary>
12     /// The end of the street.
13     /// </summary>
14     public readonly Vector2Int End;
15
16     public Street(Vector2Int start, Vector2Int end)
17     {
18         Start = start;
19         End = end;

```



```

20 }
21
22 /// <summary>
23 /// Gets a new with <see cref="Start"/> being equal to <see cref="End"/> of <c>this</c> and
24 /// vice versa.
25 /// </summary>
26 public readonly Street Flipped => new Street(End, Start);
27
28 /// <summary>
29 /// Returns the path from start to end.
30 /// </summary>
31 public readonly Vector2Int Path => End - Start;
32
33 // -- Boilerplate ausgelassen --

```

```

1 /// <summary>
2 /// Represents a <see cref="Vector2Int"/> with a direction.
3 /// </summary>
4 public readonly struct DirectedVector2Int : IEquatable<DirectedVector2Int>
5 {
6     /// <summary>
7     /// The position.
8     /// </summary>
9     public readonly Vector2Int Position;
10
11     /// <summary>
12     /// The direction.
13     /// </summary>
14     public readonly Vector2Int Direction;
15
16     // -- Boilerplate ausgelassen --
17 }

```

```

1 /// <summary>
2 /// Represents a map.
3 /// </summary>
4 public class Map
5 {
6     /// <summary>
7     /// The list of all streets.
8     /// </summary>
9     public List<Street> Streets { get; }
10
11     /// <summary>
12     /// The position of the starting point.
13     /// </summary>
14     public Vector2Int Start { get; }
15
16     /// <summary>
17     /// The position of the target.
18     /// </summary>
19     public Vector2Int End { get; }
20
21     /// <summary>
22     /// Maps the positions of intersections to all intersections directly reachable from it.
23     /// The reachable intersections are specified by their position (Target), the direction they're
24     /// in (Direction) and their distance from the other intersection (Distance).
25     /// </summary>
26     public Dictionary<Vector2Int, List<(Vector2Int Target, Vector2Int Direction, float Distance)>>
27         ReachableFromIntersection { get; }
28
29     /// <summary>
30     /// Positions of all intersections.
31     /// </summary>

```

```

30 public HashSet<Vector2Int> Intersections { get; }
31
32 /// <summary>
33 /// The smallest x and y coordinates of any intersection.
34 /// </summary>
35 public Vector2Int Min { get; }
36
37 /// <summary>
38 /// The largest x and y coordinates of any intersection.
39 /// </summary>
40 public Vector2Int Max { get; }
41
42 /// <summary>
43 /// The difference between the largest and smallest x and y coordinates of any intersection.
44 /// </summary>
45 public Vector2Int Size { get; }
46
47 public static Map FromText(string[] text)
48 {
49     if (!MapParser.TryParse(text, out var start, out var end, out var streets)) throw new
        FormatException();
50
51     return new Map(streets, start, end);
52 }
53
54 public Map(List<Street> streets, Vector2Int start, Vector2Int end)
55 {
56     Streets = streets;
57     Start = start;
58     End = end;
59     ReachableFromIntersection = new Dictionary<Vector2Int, List<(Vector2Int Target, Vector2Int
        Direction, float Distance)>>();
60     Intersections = new HashSet<Vector2Int>();
61
62     // Adds a street to the intersection at its starting point
63     void registerStreet(Street street)
64     {
65         // Get the existing list or add a new one for key street.Start
66         var reachableIntersections = ReachableFromIntersection.GetOrCreateValue(street.Start);
67
68         reachableIntersections.Add((street.End, street.Path.Direction, street.Path.Length));
69
70         Intersections.Add(street.Start);
71     }
72
73     // Add all streets to all intersections
74     foreach (var street in streets)
75     {
76         registerStreet(street);
77         // Both ends are intersections => flip to register the other
78         registerStreet(street.Flipped);
79     }
80
81     // Save these for the UI
82     Min = new Vector2Int(
83         Streets.Min(x => Math.Min(x.Start.X, x.End.X)),
84         Streets.Min(x => Math.Min(x.Start.Y, x.End.Y)));
85     Max = new Vector2Int(
86         Streets.Max(x => Math.Max(x.Start.X, x.End.X)),
87         Streets.Max(x => Math.Max(x.Start.Y, x.End.Y)));
88     Size = Max - Min;
89 }
90
91 /// <summary>
92 /// Gets the shortest path as computed by dijkstras.

```

```

93  /// </summary>
94  /// <param name="fullDistance">The length of the computed path.</param>
95  /// <returns>The path from starting point to ending point, including those points.</returns>
96  public IEnumerable<Vector2Int> ShortestPath(out float fullDistance)
97  {
98      // The position a point is reached from on the shortest currently known path leading to it.
99      var paths = new Dictionary<Vector2Int, Vector2Int>();
100     // The distance of the shortest currently known path to a given point.
101     var distances = new Dictionary<Vector2Int, float>();
102
103     var priorityQueue = new SimplePriorityQueue<Vector2Int, float>();
104
105     foreach (var intersection in Intersections)
106     {
107         var distance = intersection == Start ? 0 : float.PositiveInfinity;
108         priorityQueue.Enqueue(intersection, distance);
109         distances[intersection] = distance;
110     }
111
112     while (true) // Loop forever. Stop condition is handled by a break.
113     {
114         var head = priorityQueue.Dequeue();
115         var headDistance = distances[head];
116
117         // End is at the top of the priority queue => the path to End is the shortest unvisited
118         // path => finished, break
119         if (head == End)
120         {
121             fullDistance = headDistance;
122             break;
123         }
124
125         var reachableStreets = ReachableFromIntersection[head];
126
127         foreach (var (target, _, distance) in reachableStreets)
128         {
129             var oldDistance = distances[target];
130
131             var newDistance = headDistance + distance;
132
133             if (newDistance > oldDistance) continue;
134
135             // Update the priority only if the key already exists. Otherwise add it again.
136             if (!priorityQueue.TryUpdatePriority(target, newDistance))
137                 priorityQueue.Enqueue(target, newDistance);
138
139             paths[target] = head;
140             distances[target] = newDistance;
141         }
142     }
143
144     var path = new List<Vector2Int>();
145     for (var current = End; current != Start; current = paths[current]) path.Add(current);
146     path.Add(Start);
147     // Reverse such that Start is actually at the start and End at the end.
148     path.Reverse();
149
150     return path;
151 }
152
153 /// <summary>
154 /// For use in <see cref="BilalsPath(int, float, out int, out float)"/>.
155 /// </summary>
156 protected class Path : IComparable<Path>, IEquatable<Path>
157 {

```

```

156     public readonly int Turns;
157     public readonly float Distance;
158     public readonly Vector2Int End;
159     public readonly Vector2Int Direction;
160     public readonly Path? Previous;
161
162     public Path(int turns, float distance, Vector2Int end, Vector2Int direction, Path previous)
163     {
164         Turns = turns;
165         Distance = distance;
166         End = end;
167         Direction = direction;
168         Previous = previous;
169     }
170
171     public Path(Vector2Int end)
172     {
173         Turns = 0;
174         Distance = 0;
175         End = end;
176         Direction = default;
177         Previous = null;
178     }
179
180     /// <summary>
181     /// Returns a new <see cref="Path"/> instance with <see cref="Previous"/> set to this and
182     /// <see cref="End"/> set to <paramref name="next"/>.
183     /// </summary>
184     /// <param name="next">The <see cref="End"/> of the new <see cref="Path"/> instance.</param>
185     /// <param name="direction">The <see cref="Vector2Int.Direction"/> value of the path
186     /// between <see cref="End"/> and <paramref cref="next"/>.</param>
187     /// <param name="distance">The <see cref="Vector2Int.Length"/> value of the path between
188     /// <see cref="End"/> and <paramref cref="next"/>.</param>
189     /// <returns>The new <see cref="Path"/> instance.</returns>
190     public Path ContinueTo(Vector2Int next, Vector2Int direction, float distance)
191     {
192         var isTurn = Direction != default && Direction != direction;
193
194         return new Path(Turns + (isTurn ? 1 : 0), Distance + distance, next, direction, this);
195     }
196
197     /// <inheritdoc/>
198     public int CompareTo(Path other)
199     {
200         var turnsComp = Turns.CompareTo(other.Turns);
201         if (turnsComp != 0) return turnsComp;
202         return Distance.CompareTo(other.Distance);
203     }
204
205     /// <inheritdoc/>
206     public override bool Equals(object? obj) => obj is Path path && Equals(path);
207
208     /// <inheritdoc/>
209     public bool Equals(Path path) => End.Equals(path.End)
210         && EqualityComparer<Path?>.Default.Equals(Previous, path.Previous);
211
212     /// <inheritdoc/>
213     public override int GetHashCode() => GetHashCode.Combine(End, Previous);
214
215     public override string ToString() => (Previous == null ? string.Empty : Previous.ToString()
216         + " -> ")
217         + $"[{End}, T: {Turns}, D: {Distance}]";
218 }
219
220 /// <summary>

```

```

217  /// Gets bilals path, meaning the path with least turns shorter in length then <paramref
    name="maxLength"/>.
218  /// Returns <c>null</c> if no such path was found.
219  /// </summary>
220  /// <param name="maxLength">The longest the path can be.</param>
221  /// <param name="fullTurns">The amount of turns in the computed path.</param>
222  /// <param name="fullDistance">The length of the computed path.</param>
223  /// <returns>The path from starting point to ending point, including those points.</returns>
224  public IEnumerable<Vector2Int>? BilalsPath(int maxTurns, float maxLength, out int fullTurns,
    out float fullDistance)
225  {
226      // Path implements the priority comparisons itself => use it as key and priority type
227      var priorityQueue = new SimplePriorityQueue<Path, Path>();
228
229      // Maps intersections with approach directions to a dictionary,
230      // which contains the shortest paths to the the intersection (with appropriate direction)
231      // that is keyed by the number of turns.
232      var paths = new Dictionary<DirectedVector2Int, Dictionary<int, Path>>();
233
234      var start = new Path(Start);
235      priorityQueue.Enqueue(start, start);
236
237      Path? endPath = null;
238
239      while (priorityQueue.Any())
240      {
241          var head = priorityQueue.Dequeue();
242
243          if (head.End == End)
244          {
245              endPath = head;
246              break;
247          }
248
249          var reachableStreets = ReachableFromIntersection[head.End];
250
251          foreach (var (target, direction, distance) in reachableStreets)
252          {
253              var newPath = head.ContinueTo(target, direction, distance);
254
255              if (newPath.Distance > maxLength
256                  || newPath.Turns > maxTurns)
257              {
258                  continue;
259              }
260
261              var directedTarget = new DirectedVector2Int(target, direction);
262              var pathsToTarget = paths.GetOrCreateValue(directedTarget);
263
264              if (!pathsToTarget.TryGetValue(newPath.Turns, out var oldPath))
265              {
266                  pathsToTarget[newPath.Turns] = newPath;
267                  priorityQueue.EnqueueWithoutDuplicates(newPath, newPath);
268                  continue;
269              }
270
271              // The turns are already known to be equal => compare distances
272              if (oldPath.Distance < newPath.Distance) continue;
273
274              pathsToTarget[newPath.Turns] = newPath;
275              priorityQueue.EnqueueWithoutDuplicates(newPath, newPath);
276          }
277      }
278
279      if (endPath == null)

```

```

280     {
281         fullTurns = int.MaxValue;
282         fullDistance = float.PositiveInfinity;
283         return null;
284     }
285
286     fullTurns = endPath.Turns;
287     fullDistance = endPath.Distance;
288
289     var path = new List<Vector2Int>();
290
291     for (var current = endPath; current!.End != Start; current = current.Previous!)
292         path.Add(current.End);
293     path.Add(Start);
294     path.Reverse();
295
296     return path;
297 }
298
299 /// <summary>
300 /// Gets bilals path, meaning the path with least turns shorter in length then <paramref
301 /// name="distanceFactor"/> times the length of the shortest path.
302 /// Returns <c>null</c> if no such path was found.
303 /// </summary>
304 /// <param name="distanceFactor">The factor to apply to the length of the shortest path.</param>
305 /// <param name="shortestPath">The shortest path. Same as output of <see cref="ShortestPath(out
306 /// float)"/>.</param>
307 /// <param name="shortestPathLength">The length of the shortest path. Same as out parameter of
308 /// <see cref="ShortestPath(out float)"/>.</param>
309 /// <param name="fullTurns">The amount of turns in the computed path.</param>
310 /// <param name="fullDistance">The length of the computed path.</param>
311 /// <returns>The path from starting point to ending point, including those points.</returns>
312 public IEnumerable<Vector2Int>? BilalsPath(float distanceFactor, out IEnumerable<Vector2Int>
313     shortestPath, out int shorestPathTurns, out float shortestPathLength, out int fullTurns,
314     out float fullDistance)
315 {
316     shortestPath = ShortestPath(out shortestPathLength);
317     shorestPathTurns = shortestPath.CountTurns();
318     return BilalsPath(shorestPathTurns, distanceFactor * shortestPathLength, out fullTurns, out
319         fullDistance);
320 }
321 }

```