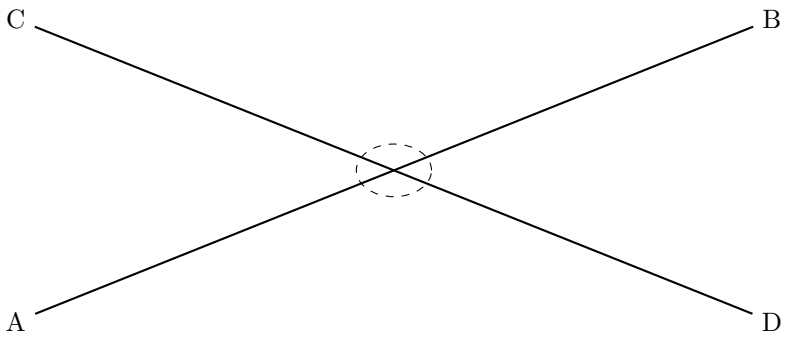
1 Lösungsidee

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.

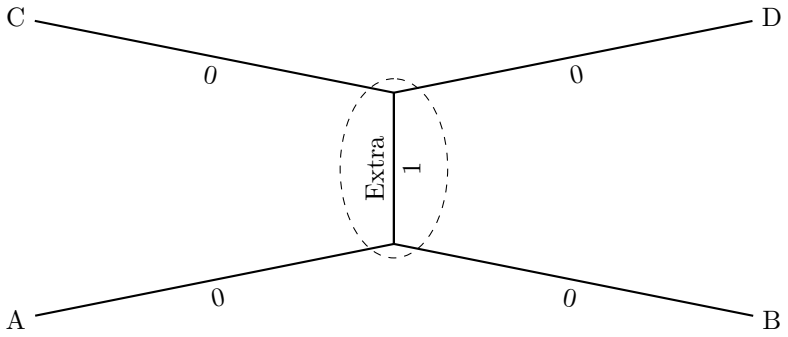
Grundlegend verläuft dieser Prozess wie der normale Dijkstra Algorithmus, jedoch mit ein paar kleinen Unterschieden: Der erste Unterschied besteht darin, dass der Algorithmus alle Wege mit einer Länge größer der Länge des kürzesten Wegs plus die erlaubte Extrastrecke (also +15% oder +30%) eliminiert.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

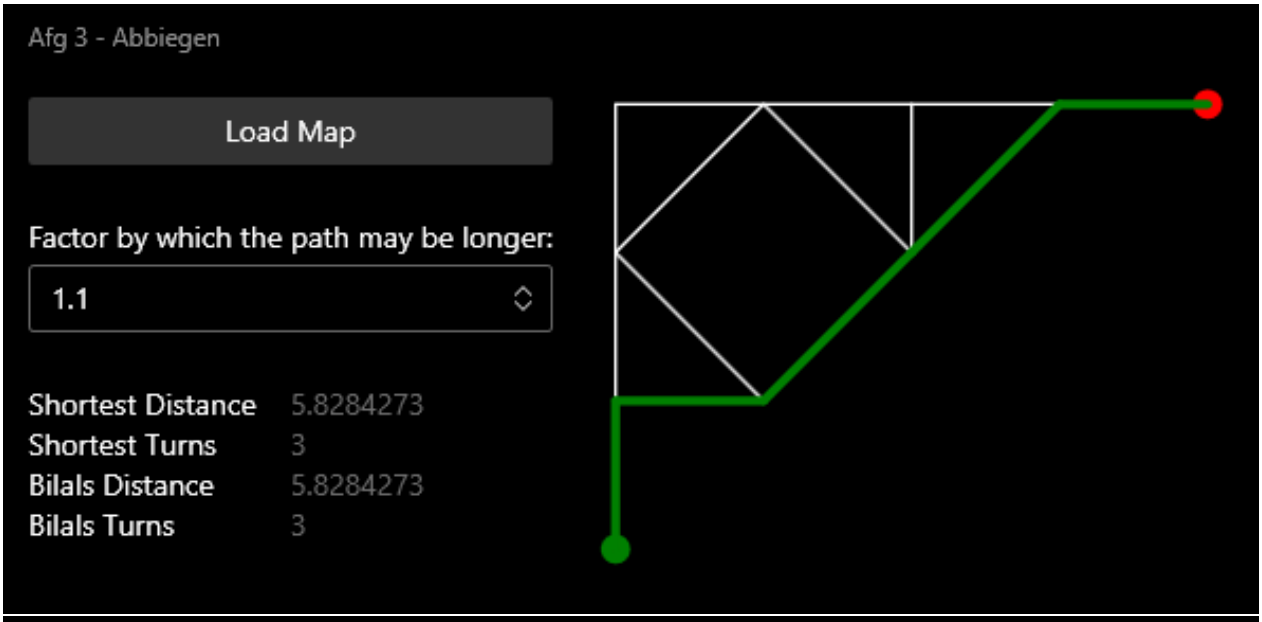
`Map` definiert dabei zwei Hauptmethoden `ShortestPath` und `BilalsPath`. `ShortestPath` ermittlet via Dijkstra den kürzesten Weg zwischen Start und Ende. `BilalsPath` ermittelt via dem obigen Algorithmus den in der Aufgabenstellung beschriebenen Weg.

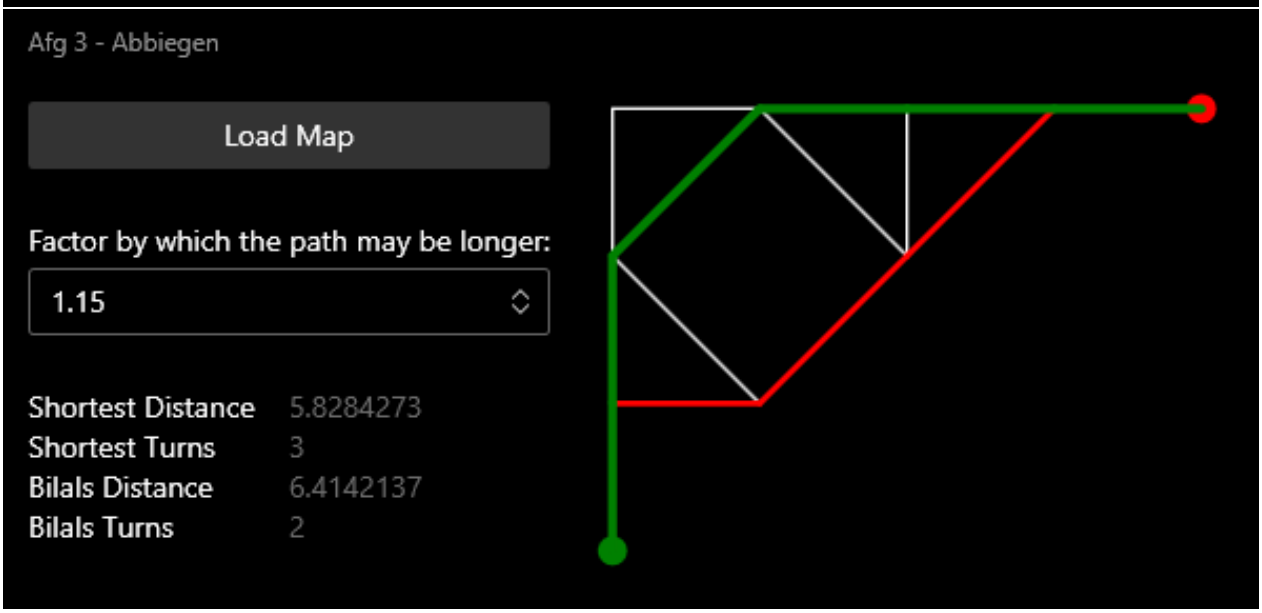
Zur Verwendung des Programms gibt es ein GUI. Zuerst muss mit “Load Map” eine der Beispieldateien geladen werden, danach kann unter “Factor by which the path may be longer” der Faktor, um den Bilals Weg länger als der kürzeste Weg sein darf eingegeben werden, also 1.15 für +15% oder 1.30 für +30%. Der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

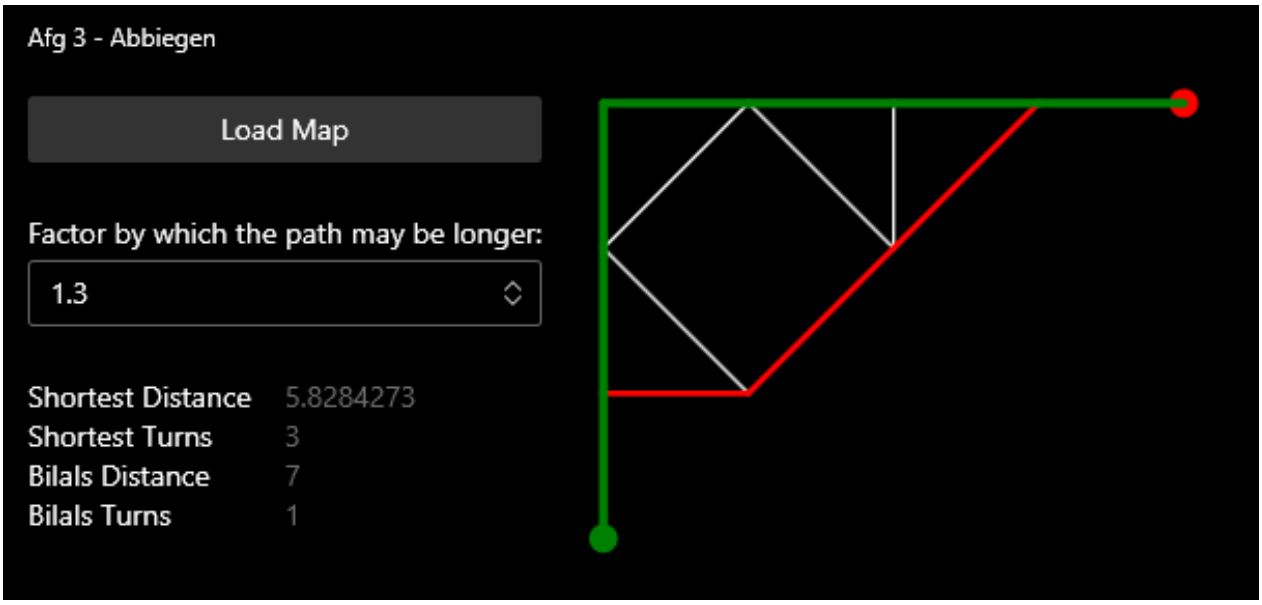
[1] https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp

3 Beispiele

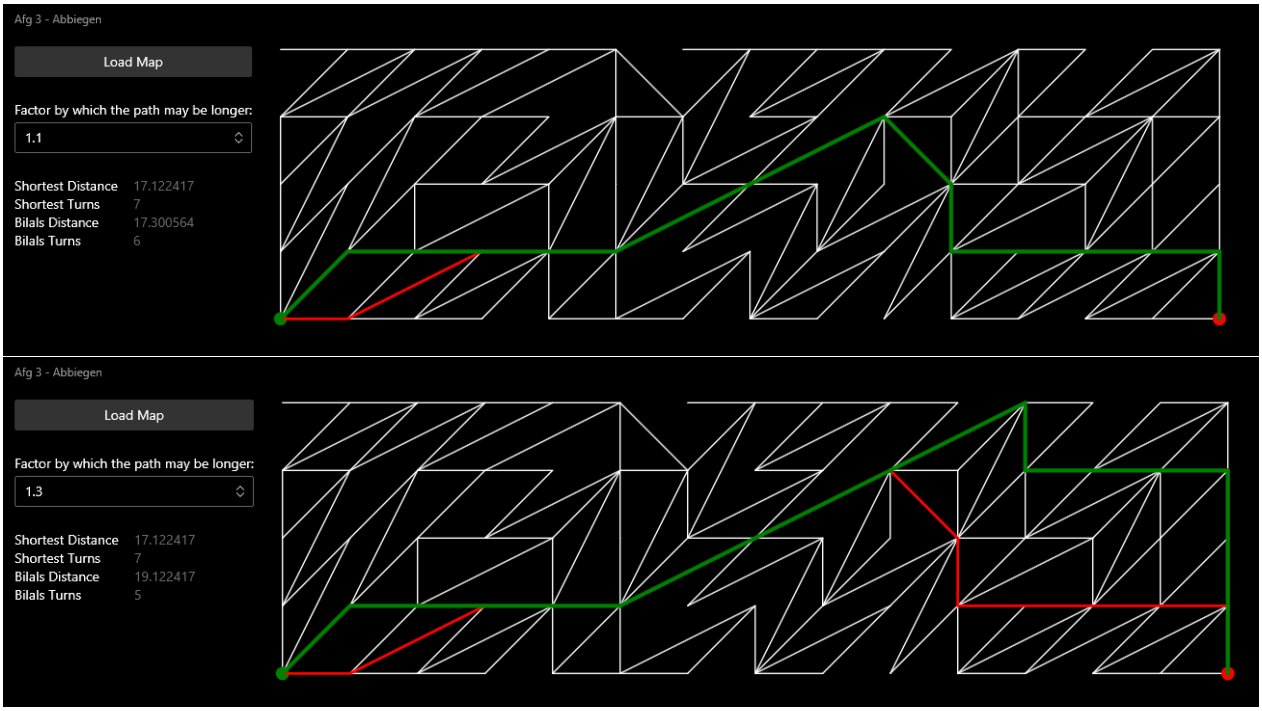
3.1



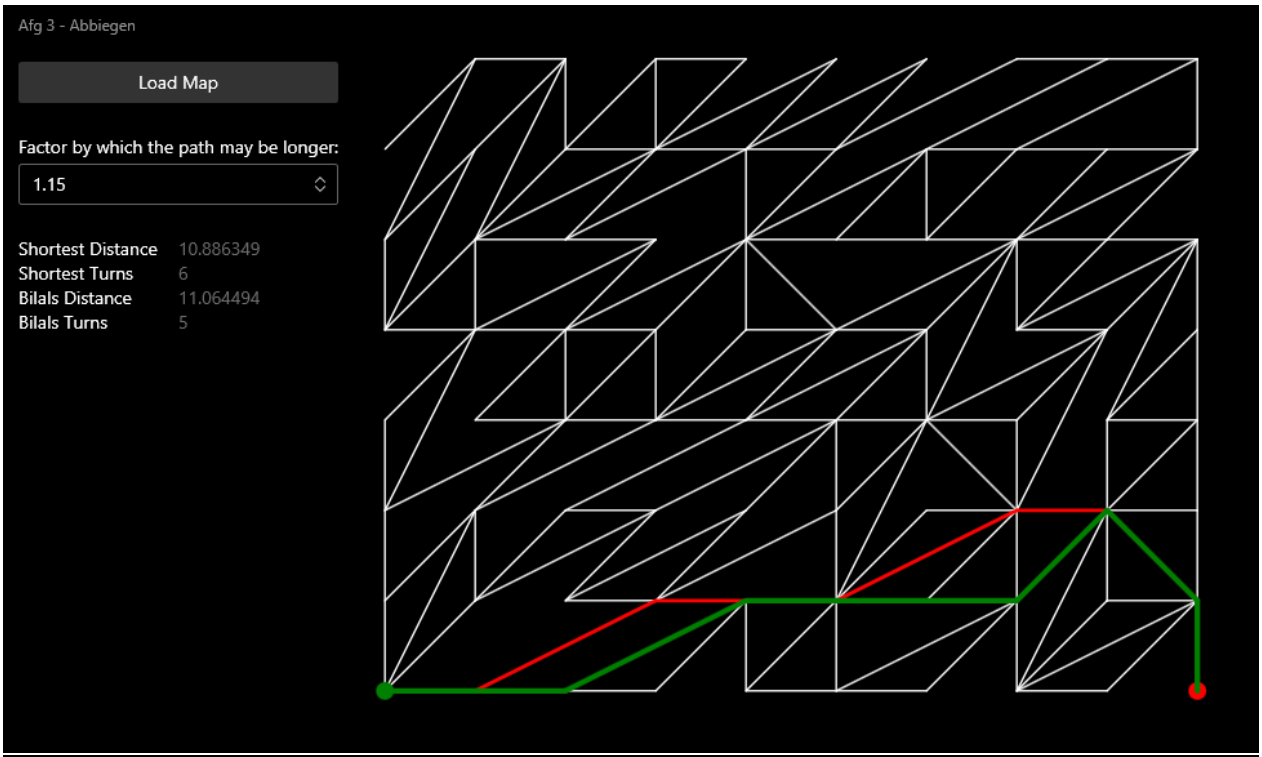


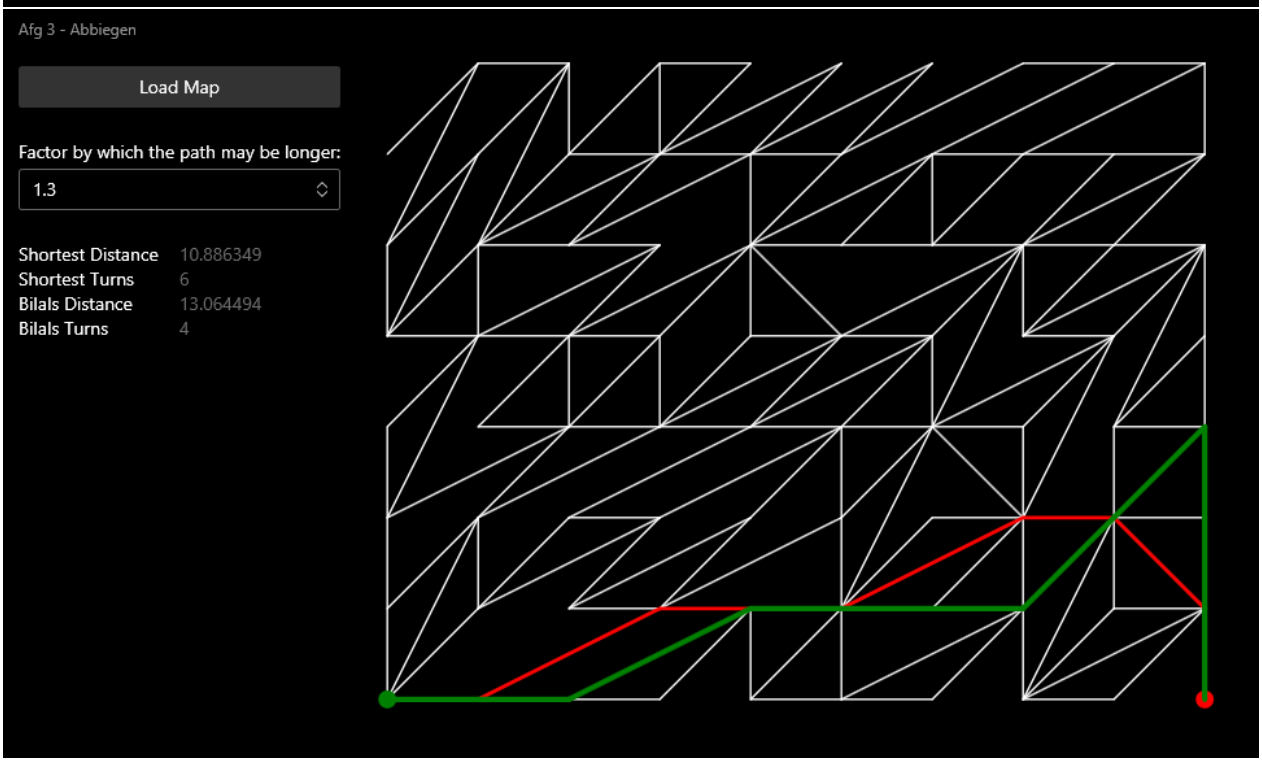

Afg 3 - Abbiegen
Load Map
Factor by which the path may be longer:
1.3
Shortest Distance 5.8284273
Shortest Turns 3
Bilals Distance 7
Bilals Turns 1

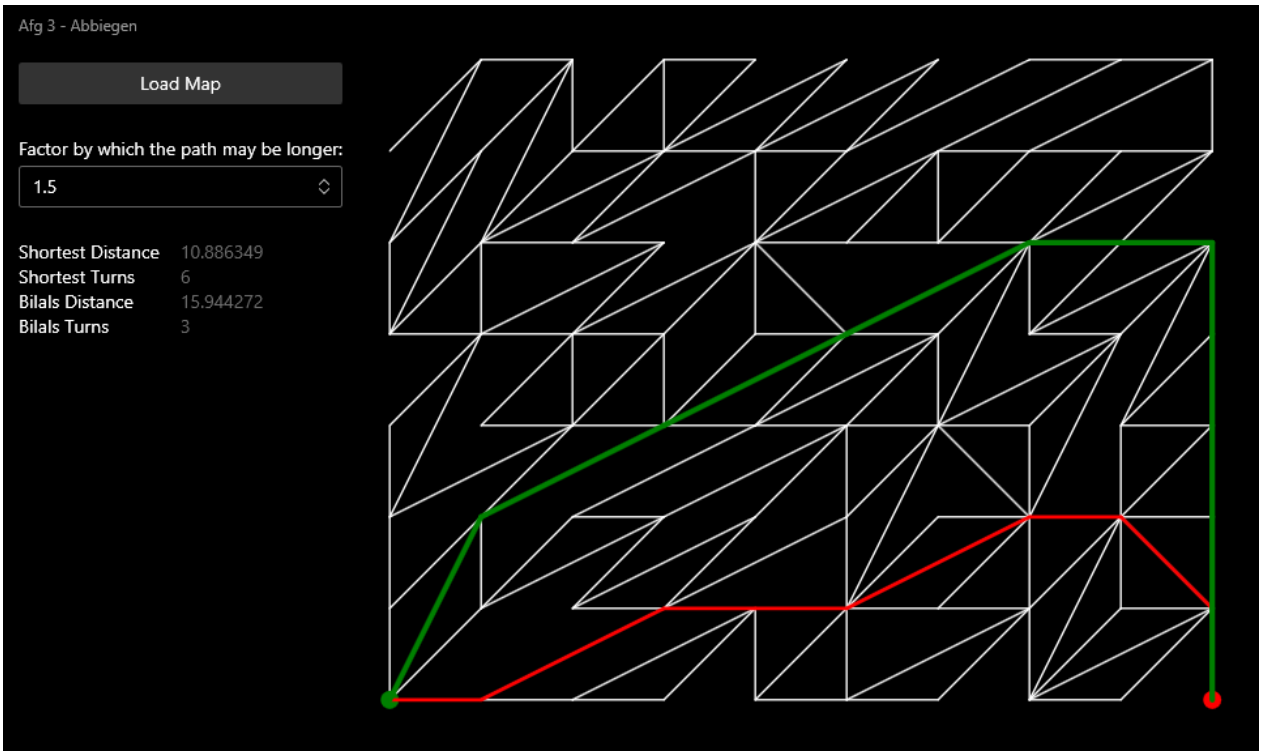
3.2


Afg 3 - Abbiegen
Load Map
Factor by which the path may be longer:
1.1
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:
1.3
Shortest Distance 17.122417
Shortest Turns 7
Bilals Distance 19.122417
Bilals Turns 5

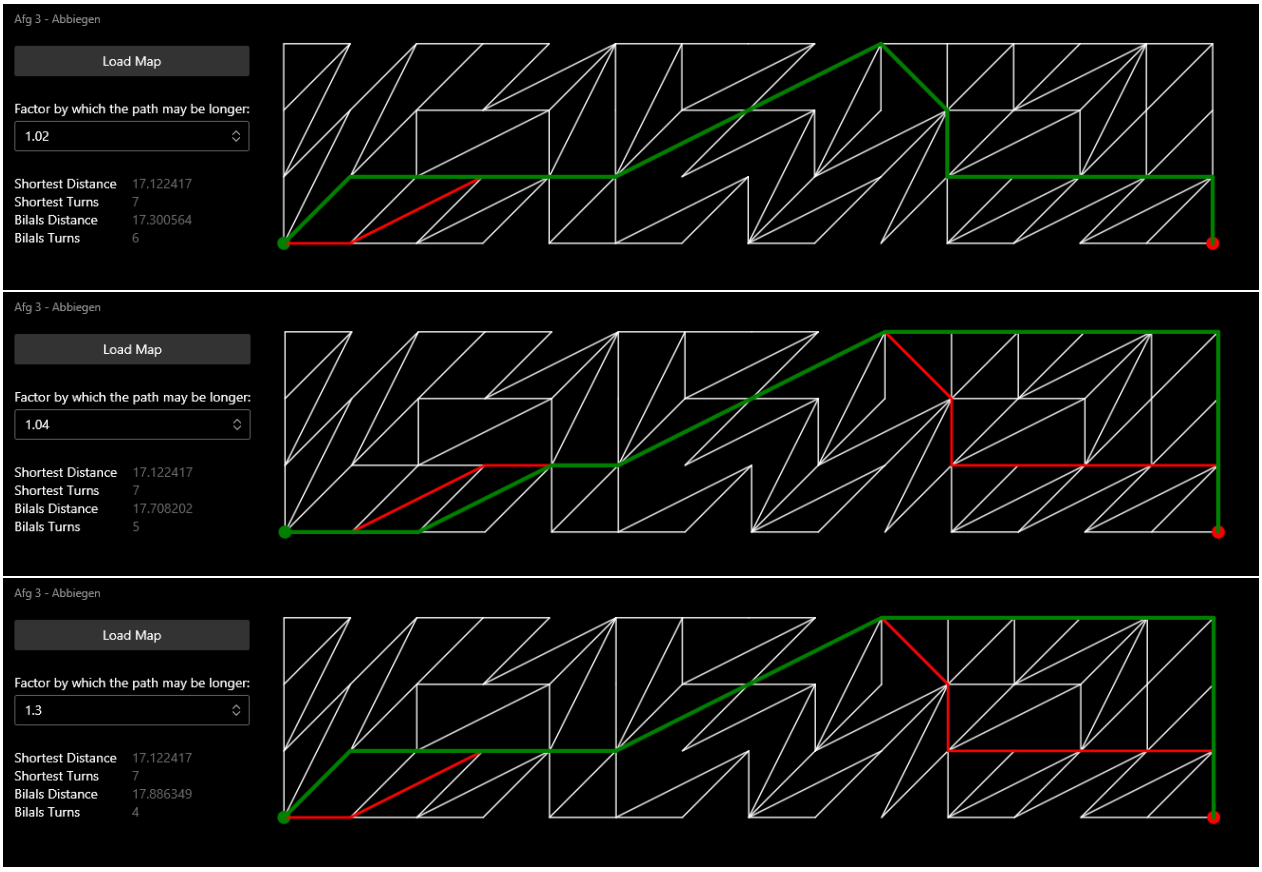
3.3


Afg 3 - Abbiegen
Load Map
Factor by which the path may be longer:
1.15
Shortest Distance 10.886349
Shortest Turns 6
Bilals Distance 11.064494
Bilals Turns 5


Afg 3 - Abbiegen
Load Map
Factor by which the path may be longer:
1.3
Shortest Distance 10.886349
Shortest Turns 6
Bilals Distance 13.064494
Bilals Turns 4


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

```
/// <summary>
/// Represents a street between two points.
/// </summary>
public readonly struct Street : IEquatable<Street>
{
    /// <summary>
    /// The start of the street.
    /// </summary>
    public readonly Vector2Int Start;

    /// <summary>
    /// The end of the street.
    /// </summary>
    public readonly Vector2Int End;

    public Street(Vector2Int start, Vector2Int end)
    {
        Start = start;
        End = end;
    }

    /// <summary>
    /// Gets a new with <see cref="Start"/> being equal to <see cref="End"/> of <c>this</c> and
        vice versa.
    /// </summary>
    public readonly Street Flipped => new Street(End, Start);

    /// <summary>
    /// Returns the path from start to end.
    /// </summary>
    public readonly Vector2Int Path => End - Start;

    // -- Boilerplate ausgelassen --
}
```

```
/// <summary>
/// Represents a <see cref="Vector2Int"/> with a direction.
/// </summary>
public readonly struct DirectedVector2Int : IEquatable<DirectedVector2Int>
{
    /// <summary>
    /// The position.
    /// </summary>
    public readonly Vector2Int Position;

    /// <summary>
    /// The direction.
    /// </summary>
    public readonly Vector2Int Direction;

    // -- Boilerplate ausgelassen --
}
```

```
/// <summary>
/// Represents a map.
/// </summary>
public class Map
{
    /// <summary>
    /// The list of all streets.
    /// </summary>
    public List<Street> Streets { get; }

    /// <summary>
    /// The position of the starting point.
```

```
    /// </summary>
    public Vector2Int Start { get; }

    /// <summary>
    /// The position of the target.
    /// </summary>
    public Vector2Int End { get; }

    /// <summary>
    /// Maps the positions of intersections to all intersections directly reachable from it.
    /// The reachable intersections are specified by their position (Target), the direction they're in (Direction) and their distance form the other intersection (Distance).
    /// </summary>
    public Dictionary<Vector2Int, List<(Vector2Int Target, Vector2Int Direction, float Distance)>> ReachableFromIntersection { get; }

    /// <summary>
    /// Positions of all intersections.
    /// </summary>
    public HashSet<Vector2Int> Intersections { get; }

    /// <summary>
    /// The smallest x and y coordinates of any intersection.
    /// </summary>
    public Vector2Int Min { get; }

    /// <summary>
    /// The largest x and y coordinates of any intersection.
    /// </summary>
    public Vector2Int Max { get; }

    /// <summary>
    /// The difference between the largest and smallest x and y coordinates of any intersection.
    /// </summary>
    public Vector2Int Size { get; }

    public static Map FromText(string[] text)
    {
        if (!MapParser.TryParse(text, out var start, out var end, out var streets)) throw new FormatException();

        return new Map(streets, start, end);
    }

    public Map(List<Street> streets, Vector2Int start, Vector2Int end)
    {
        Streets = streets;
        Start = start;
        End = end;
        ReachableFromIntersection = new Dictionary<Vector2Int, List<(Vector2Int Target, Vector2Int Direction, float Distance)>>();
        Intersections = new HashSet<Vector2Int>();

        // Adds a street to the intersection at its starting point
        void registerStreet(Street street)
        {
            // Get the existing list or add a new one for key street.Start
            var reachableIntersections = ReachableFromIntersection.GetOrCreateValue(street.Start);

            reachableIntersections.Add((street.End, street.Path.Direction, street.Path.Length));

            Intersections.Add(street.Start);
        }

        // 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 handeled 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
                    path => finished, break
118             if (head == End)
119             {
120                 fullDistance = headDistance;
121                 break;
122             }
123
124             var reachableStreets = ReachableFromIntersection[head];
125
126             foreach (var (target, _, distance) in reachableStreets)
127             {
128                 var oldDistance = distances[target];
129
130                 var newDistance = headDistance + distance;
131
132                 if (newDistance > oldDistance) continue;
133
134                 // Update the priority only if the key already exists. Otherwise add it again.
135                 if (!priorityQueue.TryUpdatePriority(target, newDistance))
                        priorityQueue.Enqueue(target, newDistance);
136
```

```
137                 paths[target] = head;
138                 distances[target] = newDistance;
139             }
140         }
141 
142         var path = new List<Vector2Int>();
143         for (var current = End; current != Start; current = paths[current]) path.Add(current);
144         path.Add(Start);
145         // Reverse such that Start is actually at the start and End at the end.
146         path.Reverse();
147 
148         return path;
149     }
150 
151     /// <summary>
152     /// For use in <see cref="BilalsPath(int, float, out int, out float)"/>.
153     /// </summary>
154     protected class Path : IComparable<Path>, IEquatable<Path>
155     {
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
                <see cref="End"/> set to <paramref name="next"/>.
182         /// </summary>
183         /// <param name="next">The <see cref="End"/> of the new <see cref="Path"/> instance.</param>
184         /// <param name="direction">The <see cref="Vector2Int.Direction"/> value of the path
                between <see cref="End"/> and <paramref cref="next"/>.</param>
185         /// <param name="distance">The <see cref="Vector2Int.Length"/> value of the path between
                <see cref="End"/> and <paramref cref="next"/>.</param>
186         /// <returns>The new <see cref="Path"/> instance.</returns>
187         public Path ContinueTo(Vector2Int next, Vector2Int direction, float distance)
188         {
189             var isTurn = Direction != default && Direction != direction;
190 
191             return new Path(Turns + (isTurn ? 1 : 0), Distance + distance, next, direction, this);
192         }
193 
194         /// <inheritdoc/>
195         public int CompareTo(Path other)
196         {
197             var turnsComp = Turns.CompareTo(other.Turns);
198             if (turnsComp != 0) return turnsComp;
```

```
199             return Distance.CompareTo(other.Distance);
200         }
201
202         /// <inheritdoc/>
203         public override bool Equals(object? obj) => obj is Path path && Equals(path);
204
205         /// <inheritdoc/>
206         public bool Equals(Path path) => End.Equals(path.End)
207             && EqualityComparer<Path?>.Default.Equals(Previous, path.Previous);
208
209         /// <inheritdoc/>
210         public override int GetHashCode() => HashCode.Combine(End, Previous);
211
212         public override string ToString() => (Previous == null ? string.Empty : Previous.ToString()
                + " -> ")
213             + $"[{End}, T: {Turns}, D: {Distance}]";
214     }
215
216     /// <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 appropiate 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!)
                 path.Add(current.End);
292         path.Add(Start);
293         path.Reverse();
294 
295         return path;
296     }
297 
298     /// <summary>
299     /// Gets bilals path, meaning the path with least turns shorter in length then <paramref
            name="distanceFactor"/> times the length of the shortest path.
300     /// Returns <c>null</c> if no such path was found.
301     /// </summary>
302     /// <param name="distanceFactor">The factor to apply to the length of the shortest path.</param>
303     /// <param name="shortestPath">The shortest path. Same as output of <see cref="ShortestPath(out
            float)"/>.</param>
304     /// <param name="shortestPathLength">The length of the shortest path. Same as out parameter of
            <see cref="ShortestPath(out float)"/>.</param>
305     /// <param name="fullTurns">The amount of turns in the computed path.</param>
306     /// <param name="fullDistance">The length of the computed path.</param>
307     /// <returns>The path from starting point to ending point, including those points.</returns>
308     public IEnumerable<Vector2Int>? BilalsPath(float distanceFactor, out IEnumerable<Vector2Int>
            shortestPath, out int shorestPathTurns, out float shortestPathLength, out int fullTurns,
            out float fullDistance)
309     {
310         shortestPath = ShortestPath(out shortestPathLength);
311         shorestPathTurns = shortestPath.CountTurns();
312         return BilalsPath(shorestPathTurns, distanceFactor * shortestPathLength, out fullTurns, out
                fullDistance);
313     }
314 }
```