Florian Pallas
pallas.florian@gmail.com
Teilnahme-ID: **51811**
Team-ID: **01171**

Aufgabe 1: Lisa rennt

37. BWINF - Runde 2

INHALT

AUFGABE[1]

Situation

Lisa möchte so spät wie möglich aufstehen, aber trotzdem ihren Bus rechtzeitig erreichen. Der Bus fährt entlang eine geraden Straße und hält für Lisa, auch wenn sie oberhalb der Haltestelle noch die Straße erreicht. Auf ihrem Weg liegen allerdings Hindernisse, die sie umlaufen muss.

Gegebene Werte

Lisa läuft mit 15km/h, der Bus fährt 30 km/h. Dieser fährt um 7:30 Uhr an der Haltestelle ab.

Im Koordinatensystem befindet sich die Haltestelle am Punkt (0, 0), die Straße verläuft entlang der y-Achse.

Die Hindernisse sind in Form von Polygonen mit deren Eckpunkten im Koordinatensystem gegeben.

Distanzen werden in Metern angegeben.

Lösung

Das Programm soll Lisa den Weg durch die Hindernisse zur Straße zeigen und berechnen, wann sie das Haus verlassen muss, um den Bus zu erreichen.

[1] Entnommen aus: 37. BWINF: 2. Runde - Aufgabenblatt (https://bwinf.de/bundeswettbewerb/37/2-runde/)

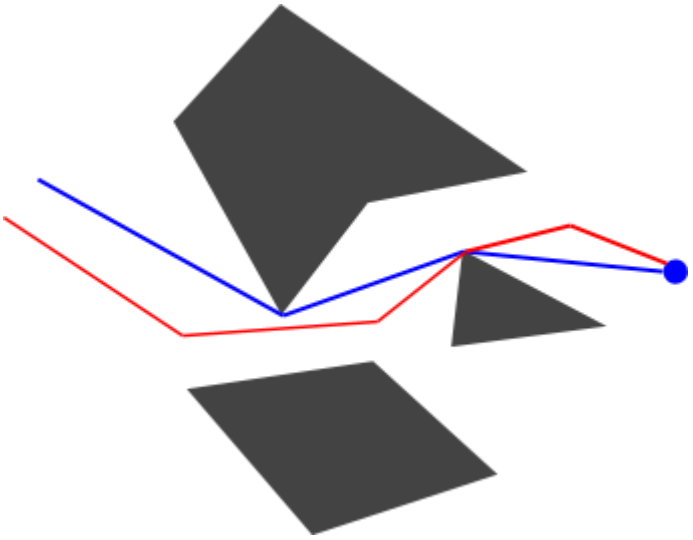
IDEE

Navigation durch Hindernisse

Um einen optimalen Weg um die Hindernisse zu finden, muss Lisa bestimmte Strecken nutzen, um weiter zu gelangen. Wenn man sich eine Skizze der Situation macht, sieht man, dass der effizienteste Weg von Eckpunkt zu Eckpunkt der Hindernisse verläuft. Einen anderen Weg einzuschlagen, würde einen Zeitverlust bedeuten.

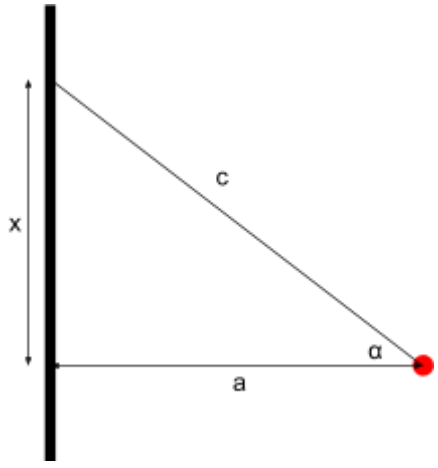
Es ist klar zu erkennen, dass der Weg von Ecke zu Ecke (blau) deutlich kürzer und damit auch schneller ist, als der alternative Weg (rot).

(Dreiecksungleichung in der Ebene)



Navigation ohne Hindernisse

Sind Lisa keine Hindernisse im Weg soll sie trotzdem den kürzesten Weg einschlagen. Zeichnet man eine Skizze, findet man heraus wie man die Strecke berechnen kann.



Da der Bus und Lisa unterschiedlich schnell sind, kann die optimale Strecke kein Lot auf der Straße sein. Es muss eine Schräge sein.

Wenn uns die Koordinaten eines Startpunktes (rot) gegeben sind, können wir die Distanz zur Straße (a) berechnen. Die Strecke entlang der Straße (x) ist unbekannt.

Die Strecke der Hypotenuse des entstandenen Dreiecks kann mithilfe Pythagoras' hier folgendermaßen ausgedrückt werden:

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

Da der Bus doppelt so schnell fährt, wie Lisa rennt, und daher früher den Einstiegspunkt erreicht, muss allerdings noch $\frac{x}{2}$ abgezogen werden. So erhält man schlussendlich den Funktionsterm:

$$f(x) = \sqrt{x^2 + a^2} - \frac{x}{2}$$

Leitet man nun ab und löst die Gleichung nach x auf, um das globale Minimum der Funktion zu bestimmen, so erhält man zwei Lösungen. Aber nur eine Lösung ist positiv und daher sinnvoll. Da die Funktion ein globales Minimum im Bereich zwischen 0 und unendlich besitzen muss und nur eine Lösung vorhanden ist, muss diese Lösung das gesuchte globale Minimum sein.

$$x = \frac{1}{\sqrt{3}} \cdot a$$

Mit dieser Beziehung kann man später, im Code, die Koordinaten des Endpunktes berechnen. Der Winkel α entspricht stets einem 30° Winkel ($tan\ \alpha = \frac{1}{\sqrt{3}}$).

Strecken

Man kann diese Ansätze nutzen, um einen Graphen aller möglichen Strecken zu erstellen. Mit dem Graph kann später der kürzeste Weg bestimmt werden.

Später wird hierzu mehr erläutert (siehe: Umsetzung - Bestimmung der Strecken).

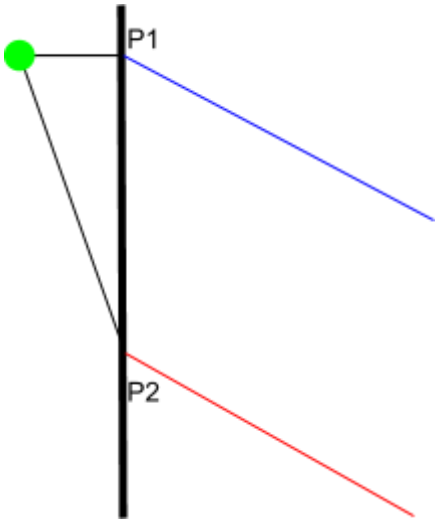
Auffinden des besten Weges im Graph

Mit diesen Berechnungen haben wir einen Graphen berechnet, welcher aus mehreren gewichteten Strecken und Knoten (Eckpunkten) besteht. Nun müssen wir allerdings noch den schnellsten Weg durch diesen finden. Dafür kann man diverse Algorithmen nutzen, am naheliegendsten ist hier jedoch Dijkstra (siehe Umsetzung: Dijkstra).

Zeitvorteil durch Höhe

Ein Aspekt wird allerdings noch ignoriert: Lisa kann Zeit gewinnen, indem sie weiter oben entlang der Straße den Bus erreicht.

Um diesen Vorteil in den Graphen zu integrieren, muss man die Gewichtung der Strecken entsprechend abändern.



Um dies umzusetzen, wird ein virtueller Endpunkt hinter der Straße gesetzt. Die Strecken zu diesem Endpunkt erhalten eine feste Gewichtung, anstatt ihrer tatsächlichen Länge. Der Weg, der die Straße am weitesten oberhalb der Haltestelle erreicht (blau), wird als Basiswert genutzt. Seine Strecke zum Endpunkt erhält die Gewichtung 0. Der Weg darunter (rot) erhält eine Zeitstrafe, da dieser früher vom Bus erreicht wird. Seine Strecke zum Endpunkt erhält folgendermaßen seine Gewichtung:

Die Streckenlänge von dem Punkt, an dem der Weg die Straße erreicht (P2), zum höchsten Punkt auf der Straße (P1), halbiert, entspricht der finalen Gewichtung.

Die Distanz zwischen beiden Punkten muss halbiert werden, denn Lisa läuft nur halb so schnell wir der Bus fährt.

Die Strecke zum Endpunkt wird nicht mit einberechnet. Sie dient lediglich dazu, den Zeitvorteil in einen Algorithmus wie Dijkstra korrekt mit einzubeziehen.

UMSETZUNG

Um meine Ideen umzusetzen nutze ich die Programmiersprache C# mit WPF in der Entwicklungsumgebung Visual Studio 2017.

Einlesen der Beispieldatei

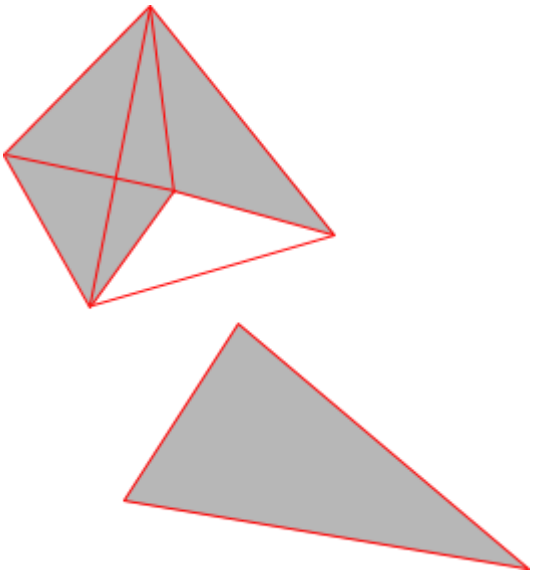
Bevor die Berechnung starten kann, muss die entsprechende Beispieldatei eingelesen werden. Wenn alle Polygone sowie der Startpunkt eingelesen wurden, werden Hindernis-Objekte erstellt. Diese halten Eckpunkte und Strecken des jeweiligen Hindernisses.

Bestimmung der Strecken

Um später einen Weg um die Hindernisse zu finden, muss zunächst definiert werden, entlang welcher Strecken der Weg verlaufen kann. Das Programm teilt diese Strecken dabei in zwei Gruppen: innere und äußere Strecken.

Innere Strecken

Innere Strecken (im Code: InnenStrecken) sind Strecken, welche die Eckpunkte eines einzigen Hindernisses verbinden.



Wie man hier sieht, werden also die Verbindungen zwischen zwei unterschiedlichen Hindernissen nicht in diese Gruppe aufgenommen. Zudem wird auch zwischen Kanten und normaler Strecken unterschieden.

Das obere Polygon besitzt beispielsweise Kanten und normale Strecken, während das Dreieck unten nur Kanten besitzt.

Diese Unterscheidung spart später bei der Kollisionsabfrage ein wenig Zeit.

```
public void SetzeInnenstrecken()
{
    // Für jedes Hindernis
    foreach(Hindernis H in Hindernisse)
    {
        List<Strecke> _Strecken = new List<Strecke>();

        foreach (Point A in H.Eckpunkte)
        {
            foreach (Point B in H.Eckpunkte)
            {
                // Gleiche Start- & Zielpunkte vermeiden
                if (A == B) { continue; }

                // Umgekehrte Strecken ignorieren (A -> B | B -> A)
                if (_Strecken.FindIndex(X => X.A == B && X.B == A) != -1) { continue; }

                // Strecke erstellen
                Strecke S = new Strecke(A, B, H);

                // Kanten bestimmen
                S.Kante = H.IstNachbar(A, B);

                // Strecke hinzufügen
                _Strecken.Add(S);
            }
        }

        // Strecken für das Hindernis setzen
        H.Strecken = _Strecken.ToArray();
    }
}
```

Die inneren Strecken werden für jedes Hindernis einzeln bestimmt. Es wird mit einer foreach-Schleife jeder Eckpunkt der Hindernisses durchlaufen und mit einer weiteren Schleife

nochmals jeder Eckpunkt durchlaufen. So erhält man für jeden Eckpunkt alle anderen Eckpunkte im Hindernis. Allerdings werden gleiche Eckpunkte abgefangen, da diese keine Strecke bilden können. Falls die neue Strecke bereits in der Gegenrichtung existiert, wird diese ebenfalls ignoriert, um keine doppelten Strecken zu erhalten.

```
public bool IstNachbar(Point A, Point B)
{
    // Indeces der Punkte bestimmen
    int IndexA = Array.IndexOf(Eckpunkte, A);
    int IndexB = Array.IndexOf(Eckpunkte, B);

    // Verbindungen vom ersten und letztem Endpunkt zählen als Kante
    if(
        (IndexA == 0 && IndexB == Eckpunkte.Length - 1) ||
        (IndexB == 0 && IndexA == Eckpunkte.Length - 1))
    { return true; }

    // Nachbar Indeces bedeuten eine Kante
    if(IndexA - IndexB == 1 || IndexA - IndexB == -1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Um zu überprüfen, ob die Strecke zwischen beiden Eckpunkten eine Kante des Polygons ist, wird überprüft, ob beide Punkte Nachbarn sind. Dabei werden die Indizes miteinander verglichen. Liegen diese hintereinander in der Liste von Eckpunkten, ist die Strecke eine Kante. Auch Verbindungen zwischen dem ersten und letzten Eckpunkt sind Kanten.

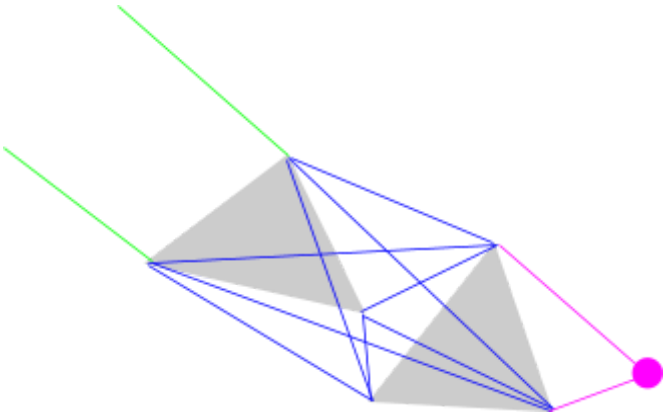
Dies ist zulässig, da im Bwlnf Forum bestätigt wurde, dass alle Polygone zulässig sind, also keine Selbstüberschrenkung aufzeigen.

Ist die Strecke eine Kante wird ein Kanten-Flag (Boolean-Wert) gesetzt, um später abfragen zu können, ob die Strecke eine Kante ist.

Schlussendlich werden die nun bestimmten Strecken dem Hindernis-Objekt übergeben.

Äußere Strecken

Äußere Strecken (im Code: AußenStrecken) sind Strecken, welche die Eckpunkte unterschiedlicher Hindernisse verbinden (blau). Zudem gelten auch die Strecken vom Startpunkt zu allen Eckpunkten (pink) und Strecken von allen Eckpunkten zur Straße (grün) als äußere Strecken.



```
public void SetzeAussenstrecken()
{
    List<Strecke> _Strecken = new List<Strecke>();

    // Strecken zwischen Hindernispunkten
    foreach (Hindernis H1 in Hindernisse)
    {
        foreach (Point A in H1.Eckpunkte)
        {
            foreach (Hindernis H2 in Hindernisse)
            {
                // Selbe Hindernisse ignorieren
                if (H1 == H2) { continue; }

                foreach (Point B in H2.Eckpunkte)
                {
                    // Umgekehrte Strecken ignorieren (A -> B | B -> A)
                    if (_Strecken.FindIndex(X => X.A == B && X.B == A) != -1) { continue; }

                    _Strecken.Add(new Strecke(A, B));
                }
            }
        }
    }
```

Um alle äußere Strecken zu finden, wird zunächst ähnlich wie bei den inneren Strecken mit zwei foreach-Schleifen jede Strecke erstellt.

```
// Strecken vom Startpunkt
foreach (Hindernis H in Hindernisse)
{
    foreach (Point B in H.Eckpunkte)
    {
        _Strecken.Add(new Strecke(Startpunkt, B));
    }
}

// Strecken im Optimalwinkel von jedem Punkt
_Strecken.Add(new Strecke(Startpunkt, ZielpunktMitOptimalwinkel(Startpunkt)));

foreach (Hindernis H in Hindernisse)
{
    foreach (Point A in H.Eckpunkte)
    {
        Point Endpunkt = ZielpunktMitOptimalwinkel(A);
        _Strecken.Add(new Strecke(A, Endpunkt) { Endstrecke = true });
    }
}
```

Die Strecken vom Startpunkt und zur Straße werden jeweils mit einer foreach-Schleife bestimmt und der Liste an Strecken hinzugefügt. Die Strecken zur Straße (im Code: Endstrecken) erhalten zusätzlich den Flag “Endstrecke”, um sie später wiederzufinden. Der Endpunkt auf der Straße wird abhängig vom Eckpunkt berechnet (siehe Idee - Navigation ohne Hindernisse).

```
private Point ZielpunktMitOptimalwinkel(Point Startpunkt)
{
    return new Point(0, Startpunkt.Y + (1 / Math.Sqrt(3) * Startpunkt.X));
}
```

Kollision von Strecken

Obwohl wir nun alle Strecken gefunden haben, sind viele von Lisa nicht begehbar. Lisa kann nur eine Strecke ablaufen, wenn diese mit keinem Hindernis kollidiert, nicht innerhalb eines Hindernisses ist oder sie sich entlang einer Kante bewegt.

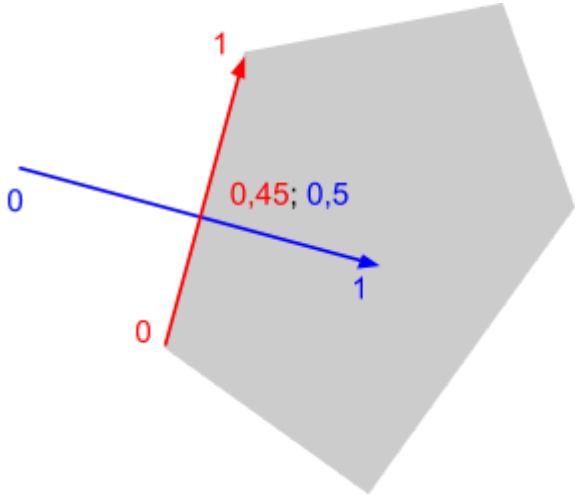
Äußere Strecken

Dafür wird nun für jede äußere Strecke überprüft, ob diese mit einem Hindernis kollidiert. Dies ist am effizientesten mit Matrizenrechnung möglich.

Diese benötigt zwei Geraden (hier: unsere Strecken) und berechnet daraus einen Vektor, welcher beschreibt ob und wo diese sich schneiden (Multiplikator für den Vektor).

```
public void TesteAussenstrecken()
{
    // Andere Strecken
    List<Strecke> _Strecken = Strecken.ToList();

    foreach (Strecke AB in Strecken)
    {
        foreach (Hindernis H in Hindernisse)
        {
            foreach(Strecke S in H.Strecken)
            {
                if(!S.Kante) { continue; }

                Vector ST;
                if (Kollision(AB, S, out ST))
                {
                    _Strecken.Remove(AB);
                }
            }
        }
    }

    Strecken = _Strecken.ToArray();
}
```

Für jede Strecke wird also jedes Hindernis mit einer foreach-Schleife durchlaufen. Nun wird die Strecke mit jeder Kante des Hindernisses auf Kollision getestet.



Sind die Parameter des Schnittpunkt Vektors aus der Matrizenrechnung zwischen 0 und 1 schneiden sich die beiden Strecken (gleich genauer).

Ist dies der Fall, wird die Strecke aus der Liste gelöscht, da sie ungültig ist.

```
private bool Kollision(Strecke SAB, Strecke SPQ, out Vector ST)
{
    // Matrix erstellen
    Vector BA = new Vector(SAB.A.X - SAB.B.X, SAB.A.Y - SAB.B.Y);
    Vector PQ = new Vector(SPQ.B.X - SPQ.A.X, SPQ.B.Y - SPQ.A.Y);
    Vector PA = new Vector(SAB.A.X - SPQ.A.X, SAB.A.Y - SPQ.A.Y);

    Matrix M = new Matrix(PQ.X, BA.X, PQ.Y, BA.Y);

    // Wenn Determinante 0 -> Keine Kollision
    if (M.Determinante() == 0.0)
    {
        ST = new Vector();
        return false;
    }

    // Schnittvektor
    Vector X = M.Invertiert().VektorMult(PA);

    double s = X.X;
    double t = X.Y;

    ST = new Vector(s, t);

    // Typ double kann zu Ungenauigkeiten führen
    const double Eps = 1E-12;

    if ((s < 0.0) || (t <= Eps) || (s > 1.0) || (t >= 1.0 - Eps)) { return false; }

    return true;
}
```

Hier kommt die Hilfsklasse “Matrix” ins Spiel, welche die notwendigen Berechnungen für diese Matrizenrechnung bereitstellt. Dazu gehört z.B. das Invertieren der Matrix, das Multiplizieren mit einem Vektor oder das Bestimmen der Determinante.

Die Determinante muss bestimmt werden, um festzustellen, ob die Gleichung eindeutig lösbar ist. Ist diese gleich 0, so ist die Gleichung nicht eindeutig lösbar und es gibt keinen eindeutigen Schnittpunkt. Also entweder gar keinen Schnittpunkt (zulässig) oder ein (teilweises) Zusammenfallen der Strecken (ebenfalls zulässig).

Die große Bedingung am Ende der Methode entscheidet ob die Strecken nun geschnitten wurden. Der Parameter s darf hier nicht den Wert 0 oder 1 annehmen, da man sonst durch Ecken das Hindernis verlassen könnte. Der Parameter t hingegen darf 0 und 1 annehmen, denn es darf durchaus in einem Eckpunkt gestartet und geendet werden. Hierbei wird aber nicht auf genau 0 oder 1 abgefragt, denn der Variablen-Typ double kann Rundungsfehler erzeugen, welche mit einer Abfrage gegen Epsilon (sehr kleiner Wert) ignoriert werden.

Strecken in Polygonen

Trotz unserer Kollisionsabfragen bisher gibt es immer noch ungültige Strecken innerhalb der Polygone (Diagonalen). Um diese auszuschließen, müssen wir weitere Kollisionen testen. Dies geschieht in mehreren Schritten:

Zunächst werden alle inneren Strecken, welche innerhalb ihres Hindernisses mit Kanten kollidieren, entfernt.

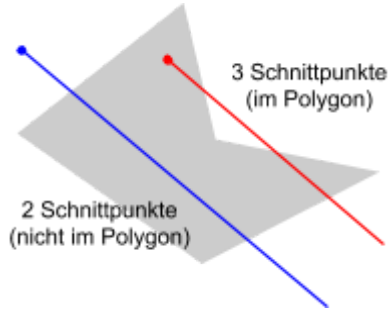
```
public void TesteInnenstreckenInnerhalb()
{
    // Alle Strecken die innerhalb eines Polygons mit einer Kante kollidieren
    foreach (Hindernis H in Hindernisse)
    {
        List<Strecke> _Strecken = H.Strecken.ToList();

        foreach (Strecke AB in H.Strecken)
        {
            // Kanten sind immer valide
            if (AB.Kante) { continue; }
            foreach (Strecke PQ in H.Strecken)
            {
                // Nur mit Kanten Kollision testen
                if (AB == PQ) { continue; }
                if (!PQ.Kante) { continue; }

                Vector ST;
                if (Kollision(AB, PQ, out ST))
                {
                    _Strecken.Remove(AB);
                }
            }
        }

        H.Strecken = _Strecken.ToArray();
    }
}
```

Dies geschieht erneut mit einer foreach-Schleife. Dabei werden Kanten ignoriert, da diese nicht mit anderen Kanten kollidieren.

Diese Strecken werden dann gegen alle Kanten im Hindernis auf Kollision abgefragt und dementsprechend entfernt.

Selbige Strecken werden im Anschluß, noch einmal ähnlich auf Kollision mit Kanten aus anderen Hindernissen getestet.

Als letzte Maßnahme wird der Punkt-in-Polygon-Test nach Jordan[2] durchgeführt. Dieser Test macht sich zunutze, dass die Anzahl der Schnittpunkte einer am Punkt startenden Halbgeraden mit einem Polygon nur dann gerade ist, wenn der Punkt, von dem die Halbgerade ausgeht, außerhalb des Polygons ist.



Der rote Punkt ist im Polygon, da seine Halbgerade das Polygon 3 mal (ungerade Anzahl) schneidet.

Der blaue Punkt ist außerhalb, da seine Halbgerade das Polygon 2 mal (gerade Anzahl) schneidet.

Die Richtung der Halbgerade ist dabei frei wählbar.

[2] Implementiert nach: Wikipedia - Punkt-In-Polygon-Test nach Jordan

```
public void PunktInPolygon()
{
    // Punkt-in-Polygon-Test nach Jordan
    const double Epsilon = 0.000001;
    const double Laenge = 100000;
    Random R = new Random();

    foreach (Hindernis H in Hindernisse)
    {
        List<Strecke> _Strecken = H.Strecken.ToList();

        foreach(Strecke AB in H.Strecken)
        {
            // Kanten sind valide
            if(AB.Kante) { continue; }

            int Anzahl = 0;

            // Startpunkt - Epsilon * Vektor von Punkt aus
            Point X = AB.A + Epsilon * AB.Vektor();

            // Richtung bestimmen
            Vector Richtung;
            Vector VAB = AB.Vektor();
            Vector VBA = new Vector(VAB.Y, VAB.X);

            Richtung = new Vector(1, 0);
            bool IstGueltig;

            // Richtung so lange variieren bis sie gültig ist
            while (true)
            {
                IstGueltig = true;

                foreach (Strecke PQ in H.Strecken)
                {
                    Vector VPQ = PQ.B - PQ.A;
                    Vector VQP = PQ.A - PQ.B;

                    // Wenn parallel
                    if (Vector.Determinant(Richtung, VPQ) == 0 || Vector.Determinant(Richtung, VQP) == 0)
                    {
                        IstGueltig = false;
                        break;
                    }
                }

                // Abbrechen wenn gültig
                if (!IstGueltig)
                {
                    Richtung = new Vector(R.NextDouble(), R.NextDouble());
                    continue;
                }
                else
                {
                    break;
                }
            }
```

Zuerst wird der Startpunkt der Halbgeraden bestimmt. Der Startpunkt unserer Strecke kann allerdings nicht sofort verwendet werden. Um zu garantieren, dass kein Fehler bei der Berechnung entsteht, wird der Punkt ein Epsilon zum inneren der Strecke bewegt.

Auch die Richtung der Halbgeraden muss erst einmal bestimmt werden. Dabei muss beachtet werden, dass die Richtung nicht parallel zu einer der andern Strecken sein darf, da sonst eventuell unendlich viele Schnittpunkte existieren. Daher wird vorher überprüft, ob eine

Strecke parallel zur Halbgeraden ist. Ist dies der Fall, wird die Richtung zufällig geändert und der Prozess wiederholt, bis eine gültige Halbgerade gefunden wurde.

```
        // Strahlstrecke erstellen
        Strecke Strahl = new Strecke(X, X + Richtung * Laenge);

        // Kollision testen
        foreach (Strecke PQ in H.Strecken)
        {
            if (!PQ.Kante) { continue; }
            if (AB == PQ) { continue; }

            Vector ST;
            if (Kollision(Strahl, PQ, out ST))
            {
                Anzahl++;
            }
        }

        // Wenn die Anzahl and Treffern nicht gerade ist -> im Polygon
        if (Anzahl % 2 != 0)
        {
            _Strecken.Remove(AB);
        }
    }

    H.Strecken = _Strecken.ToArray();
}
```

Nun kann die Halbgerade (im Code: Strahl) konstruiert werden. Wir benutzen für die Kollisionsabfrage dieselbe Methode wie zuvor. Ist die Anzahl ungerade, so wird die Strecke entfernt.

Auffinden des besten Weges im Graph

Endstrecken

Der Graph ist nun komplett, allerdings haben wir den Zeitvorteil[3] noch nicht miteinberechnet.

```
public void SetzeEndstrecken()
{
    List<Strecke> _Strecken = Strecken.ToList();

    // Höchsten Endpunkt bestimmen
    double HoechsterEndpunkt = double.NegativeInfinity;
    foreach (Strecke S in Strecken)
    {
        if (!S.Endstrecke) { continue; }

        if(S.B.Y >= HoechsterEndpunkt)
        {
            HoechsterEndpunkt = S.B.Y;
        }
    }

    // Festen Betrag setzen
    foreach (Strecke S in Strecken)
    {
        if(!S.Endstrecke) { continue; }

        Endpunkt = new Point(-10, 0);
        Strecke SEndpunkt = new Strecke(S.B, Endpunkt);
        SEndpunkt.HatFestenBetrag = true;
        SEndpunkt.FesterBetrag = (HoechsterEndpunkt - S.B.Y) / 2;

        _Strecken.Add(SEndpunkt);
    }

    Strecken = _Strecken.ToArray();
}
```

Dazu wird der höchste Endpunkt (Punkt auf Straße) bestimmt. Nun wir die Gewichtung (hier: Betrag) von allen anderen Strecken gesetzt.

Diese entspricht der Differenz der Y-Koordinate des höchsten Endpunktes und des zu bearbeitenden Punktes, halbiert.

Dijkstra

Der Graph ist nun bereit bearbeitet zu werden. Das Programm verwendet dafür den Algorithmus Dijkstra[4]. Der Dijkstra-Algorithmus ist der Standardalgorithmus, um in einem solchen Graphen die kürzeste und damit schnellste Verbindung zu finden. Zudem bietet er das bestmögliche Laufzeitverhalten.

Der Algorithmus durchläuft den Graphen, welcher aus Knoten und Kanten besteht. Ein Start- und Endknoten ist gegeben.

Ist der Algorithmus fertig, kann man vom Endknoten aus zurück zum Startknoten finden und dabei die durchlaufenen Knoten notieren. Man erhält eine Reihe von Knoten, welche den kürzesten und damit schnellsten Weg darstellen.

---

[3] Siehe: Idee - Zeitvorteil durch Höhe
[4] Implementiert nach: Wikipedia - Dijkstra-Algorithmus

Genauer werde ich den Dijkstra-Algorithmus nicht erklären.

Aus den Strecken, welche wir berechnet haben, wird nun ein Graph mit Knoten und gewichteten Kanten generiert. Die Gewichtung stellt den Betrag der Strecken dar.

```
private void Setup()
{
    // Alle Vertices erstellen
    foreach(Strecke S in Strecken)
    {
        // Vertex nur hinzufügen wenn sie nicht schon existiert
        if(!Vertices.Exists(x => x.Pos == S.A))
        {
            Vertices.Add(new DijkstraVertex() { Pos = S.A });
        }

        if (!Vertices.Exists(x => x.Pos == S.B))
        {
            Vertices.Add(new DijkstraVertex() { Pos = S.B });
        }
    }

    // Nachbarn setzen
    foreach (DijkstraVertex V in Vertices)
    {
        List<DijkstraVertex> Nachbarn = new List<DijkstraVertex>();
        List<double> NachbarWerte = new List<double>();

        // Strecken finden die mit diesem Knoten verbunden sind (Am Punkt A)
        List<Strecke> NachbarStreckenA = Strecken.FindAll(x => x.A == V.Pos);
        foreach(Strecke S in NachbarStreckenA)
        {
            // Nachbarknoten setzen
            Nachbarn.Add(Vertices.Find(x => x.Pos == S.B));
            // Weg zu Nachbarknoten setzen
            NachbarWerte.Add(S.Betrag());
        }

        // Strecken finden die mit diesem Knoten verbunden sind (Am Punkt B)
        List<Strecke> NachbarStreckenB = Strecken.FindAll(x => x.B == V.Pos);
        foreach (Strecke S in NachbarStreckenB)
        {
            // Nachbarknoten setzen
            Nachbarn.Add(Vertices.Find(x => x.Pos == S.A));
            // Weg zu Nachbarknoten setzen
            NachbarWerte.Add(S.Betrag());
        }

        // Listen setzen
        V.Nachbarn = Nachbarn;
        V.NachbarWerte = NachbarWerte;
    }

    // Start- und Endvertex bestimmen
    Startvertex = Vertices.Find(x => x.Pos == Startpunkt);
    Endvertex = Vertices.Find(x => x.Pos == Endpunkt);
}
```

Jeder Start- und Endpunkt einer Strecke stellt einen Knoten (eine Vertex) dar. Kanten werden in dieser Implementierung von Dijkstra nicht als eigene Objekte umgesetzt. Es existiert lediglich eine Liste für jeden Knoten, welche alle Beträge (bzw. Gewichtungen) für den Weg zu den jeweiligen Nachbarknoten halten (effizienter).

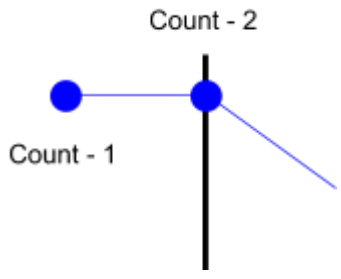
Berechnen der Zeit

Nachdem wir den Dijkstra-Algorithmus angewendet haben, wissen wir nun, welche Knoten den schnellsten Weg darstellen und wie lang dieser ist. Zudem wissen wir, an welchem Punkt entlang der Straße Lisa den Bus erreicht. Daraus können wir nun die Zeit berechnen, zu der Lisa loslaufen muss.

Ich benutze dafür die Typen “DateTime” und “TimeSpan” von C# um Rechenfehler zu vermeiden.

```
DateTime BusStartzeit = new DateTime(2019, 4, 1, 7, 30, 0);

double BusStreckeKm = Sequenz[Sequenz.Length - 2].Pos.Y / 1000;
double BusFahrtzeitStunden = BusStreckeKm / 30.0;
double BusFahrtzeitSekunden = BusFahrtzeitStunden * 60 * 60;

double LisaStreckeM = Sequenz[Sequenz.Length - 2].Distanz;
double LisaStreckeKm = LisaStreckeM / 1000;
double LisaLaufzeitStunden = LisaStreckeKm / 15.0;
double LisaLaufzeitSekunden = LisaLaufzeitStunden * 60 * 60;
TimeSpan LisaLaufzeit = new TimeSpan(0, 0, 0, (int)Math.Round(LisaLaufzeitSekunden));

DateTime BusEinstieg = BusStartzeit.AddSeconds(BusFahrtzeitSekunden);

DateTime LisaStartzeit = BusEinstieg.Subtract(LisaLaufzeit);

int StreckeInt = (int)Math.Round(LisaStreckeM);
```

“Sequenz[Sequenz.Length - 2]” ist hier der Knoten, welcher an der Straße anliegt. Der virtuelle Endknoten wird also ignoriert:



Der Endknoten besitzt den Index “Sequenz.Count - 1”, da Indizes in Arrays null-basierend sind. Der Punkt auf der Straße entspricht dementsprechend dem Index “Sequenz.Count - 2”.

Zuerst berechnen wir, wie lang der Bus benötigt, bis er dort angekommen ist, wo Lisa einsteigt. Diesen Wert rechnen wir in Sekunden um und addieren ihn auf die Abfahrtszeit (7:30 Uhr) des Busses. Als nächsten Schritt berechnen wir, wie lange Lisa für ihren Weg benötigt. Diesen Wert rechnen wir erneut in Sekunden aus und ziehen diesen von der Zeit, an der Lisa einsteigt, ab. Nun wissen wir, wann Lisa anfangen muss loszulaufen.

Die Strecke und die Zeit wird auf Meter und Sekunden gerundet ausgegeben.

Komplexität

Um die Komplexität zu bestimmen, muss man das Programm in seine Einzelschritte unterteilen:

Nehmen wir an, es gibt N Eckpunkte und ein Hindernis besitzt durchschnittlich 5 Eckpunkte.

Im 1. Teil des Programms werden die möglichen Strecken bestimmt, die Lisa ablaufen kann. Dazu werden alle Eckpunkte mit allen anderen Eckpunkten verbunden (N * N = N^2 Operation). Dazu werden noch die Strecken vom Startpunkt zu allen Ecken und die Strecken von allen Ecken zur Straße bestimmt (2N Operation). Die Kanten werden nun auf Kollision geprüft. Dabei wird zuerst jede Kante auf Kollision mit allen anderen Kanten geprüft (N^2 * N = N^3 Operation). Der Punkt-In-Polygon-Test ist im Vergleich harmlos. Er vergleicht nur durchschnittlich 5 mal 5N Kanten (25N Operation).

1. Teil insgesamt: $O(N) = N^3$

Im 2. Teil des Programms wird Dijkstra auf den Graphen angewendet. Mit einer sortierten Liste erreicht dieser eine Komplexität von $O(N) = N\ log\ N$. Allerdings ist für diesen Anwendungszweck auch eine herkömmlich Liste akzeptabel. Hier, in meinem Programm, entspricht der Dijkstra-Algorithmus also einer N^2 Operation.

2. Teil insgesamt: $O(N) = N^2$

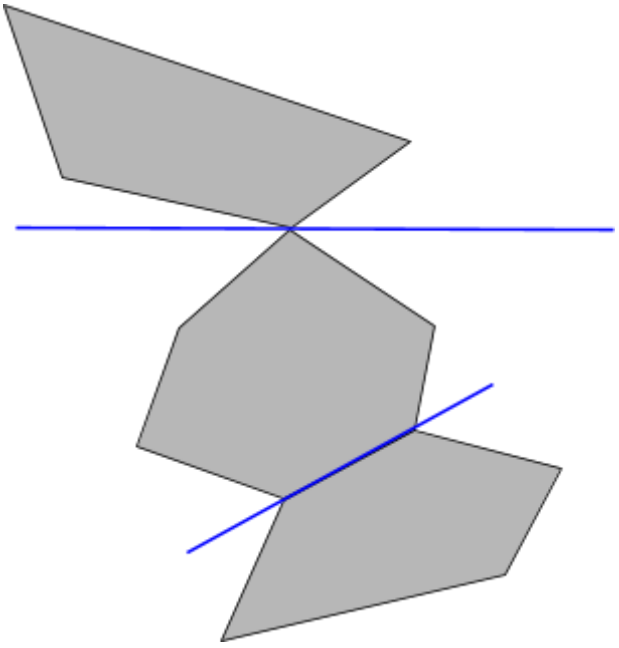
Insgesamt erhält man folgende Komplexität: $O(N) = N^3$

Diese Komplexität ist harmlos im Falle der Beispielaufgaben, kann aber, wenn nötig noch verbessert werden.

Qualität der Lösung & Verbesserungen

Da die Lösung rein mathematisch ermittelt wurde und Dijkstra den tatsächlich kürzesten Weg findet, stellt meine Lösung die beste Lösung dar. Alle Möglichkeiten wurden einbezogen.

Trotzdem sind Verbesserungen möglich. Es gibt Situationen in denen mein Programm nicht korrekt handelt:



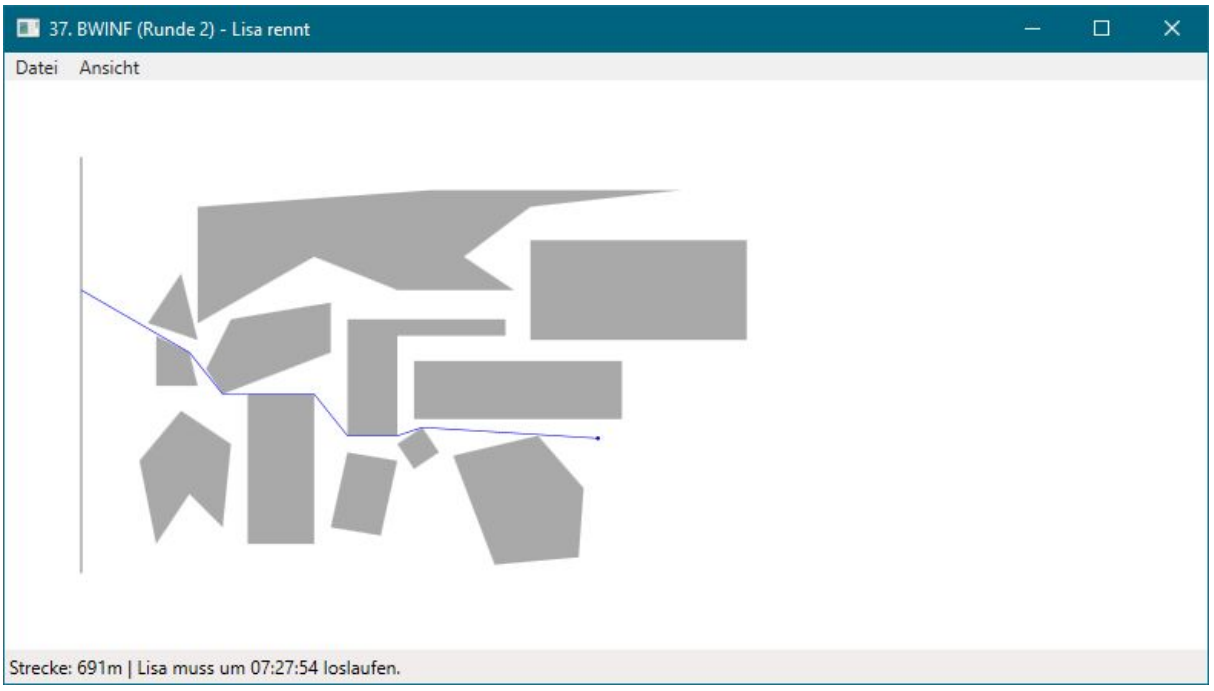
In diesen Situationen, sollte es Lisa nicht möglich sein die Hindernisse zu durchgehen. Da dieser Fall weder explizit erwähnt, noch in der Beispielen aufgetaucht ist, erkennt mein Programm solche Situationen nicht und lässt Lisa passieren.

Dies würde weitere Abfragen in der Kollisions-Methode benötigen.

BEDIENUNG

Das Programm ist mit seiner visuellen Oberfläche leicht zu bedienen. Um eine Beispieldatei zu öffnen, klickt man in der Menüleiste (oben) auf “Datei” und danach im Kontextmenü auf “Öffnen”. Im Dateidialog wählt man nun die gewünschte Beispieldatei (*.txt).

Die Berechnung startet sofort.



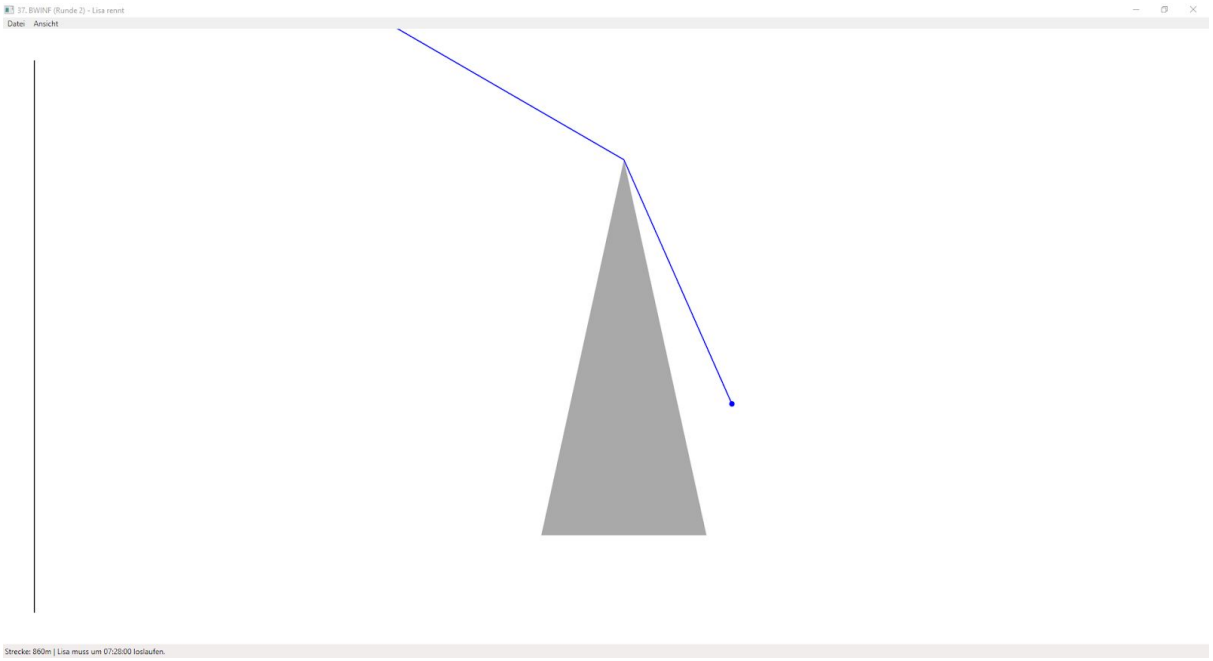
In der Mitte des Fensters werden nun die Hindernisse (grau), die Straße (links) als auch Lisas Startpunkt (blauer Punkt) und der beste Weg (blau) angezeigt. In der Statusleiste (unten) wird die Strecke, die Lisa zurücklegen muss und die Zeit, zu der sie loslaufen muss, angezeigt.

Im Kontextmenü “Ansicht” können zudem verschiedene andere Dinge, wie die inneren oder äußere Strecken an- und ausgeschaltet werden. Dies zeigt noch einmal anschaulich, wie der Algorithmus die Strecken bestimmt und nutzt.

ERGEBNISSE

Aufgabe 1:

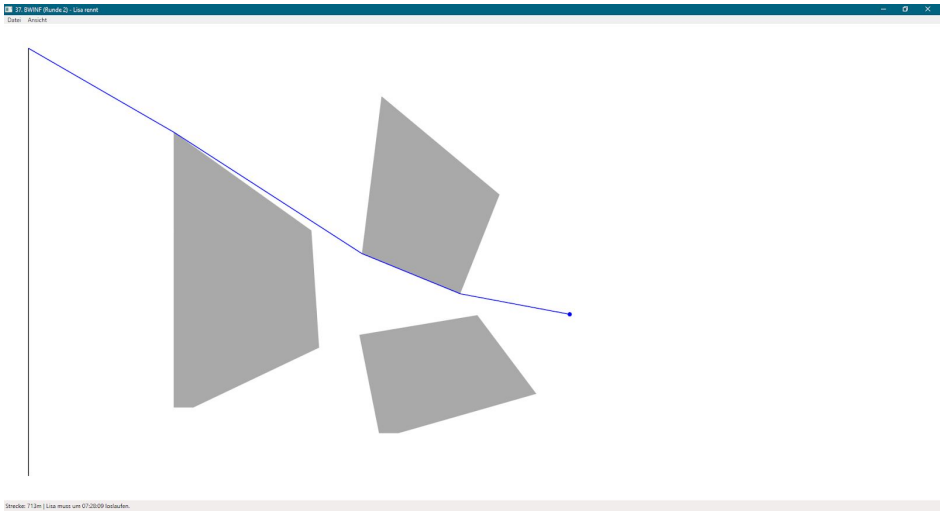
Bei der 1. Beispielaufgabe wählt der Algorithmus den oberen Weg, um den Zeitvorteil in der Höhe zu nutzen.



Strecke: 860 m | Lisa muss um 07:20:00 Uhr loslaufen.

Aufgabe 2:

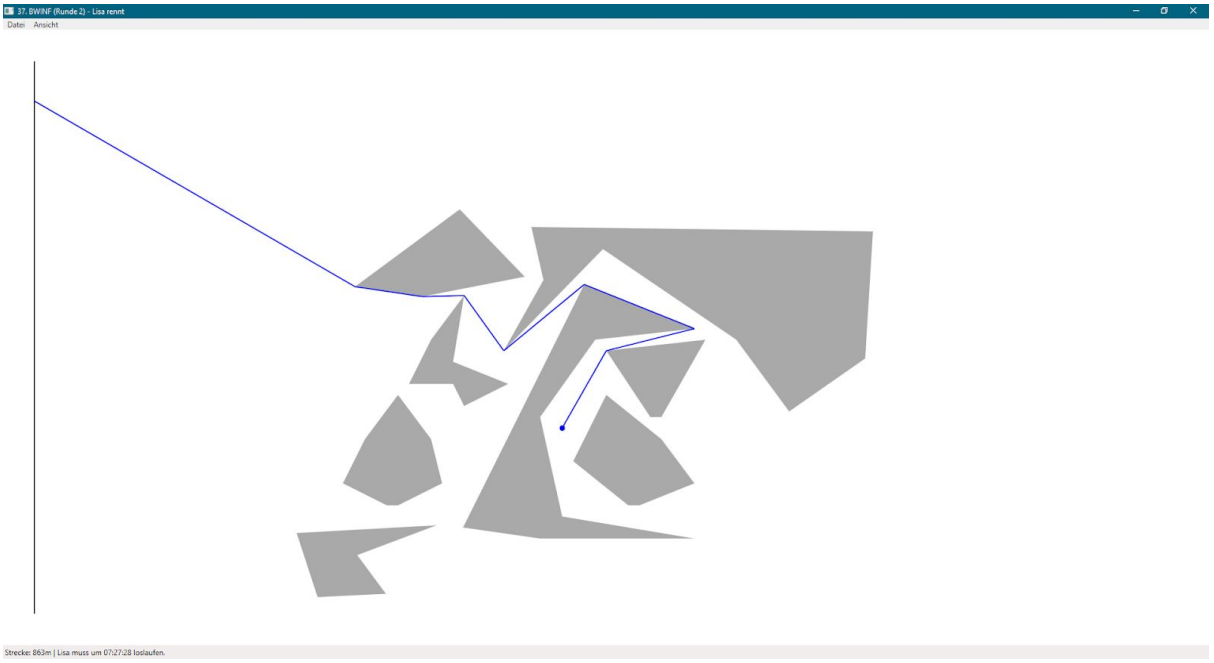
Bei der 2. Aufgabe führt der schnellste Weg durch die Hindernisse. Trotz des Zeitvorteils entscheidet sich der Algorithmus nicht oberhalb der Hindernisse entlang zu gehen, denn der Weg scheint den Vorteil nicht Wert zu sein.



Strecke: 713 m | Lisa muss um 07:28:09 Uhr loslaufen.

Aufgabe 3:

Auch hier tendiert der Algorithmus nach oben zu gehen, allerdings nicht zu hoch, da es sich nicht mehr lohnt.



Strecke: 863 m | Lisa muss um 07:27:28 Uhr loslaufen.

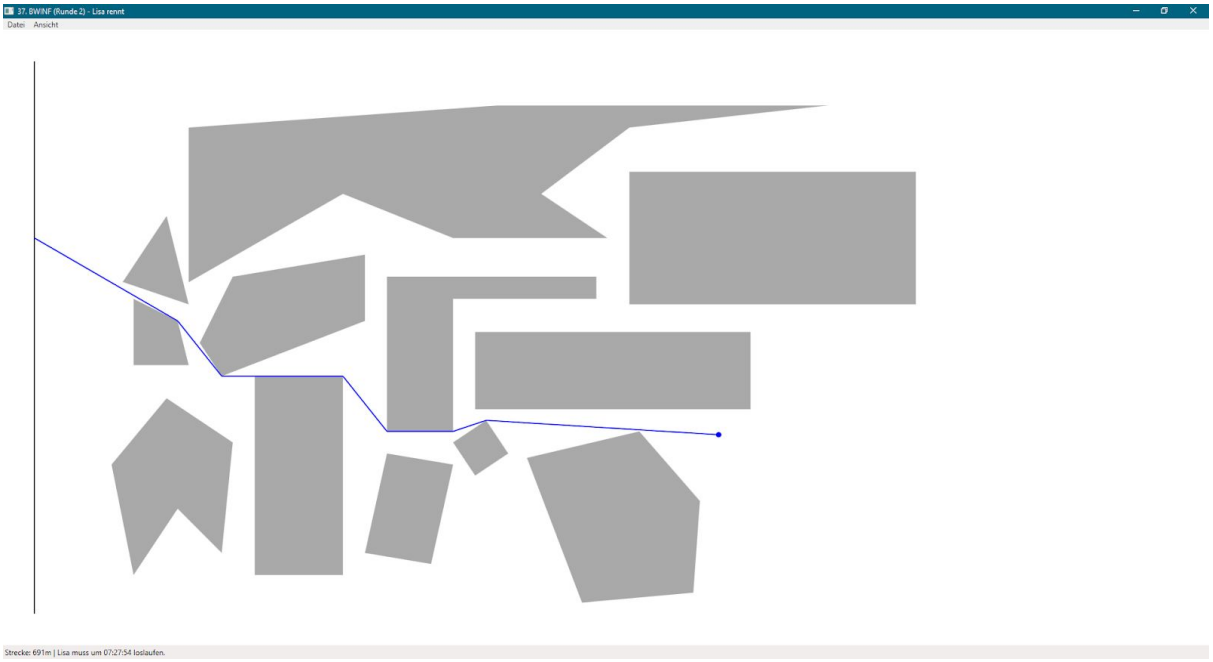
Aufgabe 4:

Hier ist es schneller, über die Hindernisse hinweg zu gehen. Man beachte auch, dass der Weg am ersten Hindernis knapp entlang der Kante verläuft, um Zeit zu sparen.

Strecke: 1263 m | Lisa muss um 07:26:56 Uhr loslaufen.

Aufgabe 5:

Da der Weg nach oben durch große Hindernisse blockiert ist, ist der Weg durch die Mitte deutlich schneller. Oben entlang zu gehen, würde einen großen Umweg und daher Zeitverlust bedeuten.



Strecke: 691 m | Lisa muss um 07:27:54 Uhr loslaufen.