Aufgabe 1: Lisa rennt

Team-ID: 00922

Team-Name: Zweiundvierzig

Bearbeiter/-innen dieser Aufgabe:
Franz Miltz

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabe und meine Lösung umfassen mehrere Teilprobleme. Das Erste, was sich stellt, ist der triviale Fall. Hierzu muss bestimmt werden, in welchem Winkel Lisa laufen muss, wenn sie kein Hindernis in ihrem Weg hat. Dass sie in diesem Fall einer geraden Strecke folgt ist klar, da anderenfalls ein kürzerer Weg zum Schnittpunkt mit der Straße existieren würde.

Um die komplizierteren Fälle zu betrachten, werden zwei wesentliche Schritte benötigt. Zunächst müssen alle *relevanten* Punkte betrachtet werden, die für die Ermittlung des optimalen Weges eine Rolle spielen. Wenn diese Punkte bestimmt sind, müssen in den resultierenden Graphen alle möglichen Kanten (optimalerweise mit Kantenlängen) eingezeichnet werden. In diesem Graphen muss nun der Pfad gesucht werden, den Lisa gehen muss, um so spät wie möglich ihr Haus verlassen zu können.

1.1 Freier Weg

Wir betrachten das Dreieck ΔABC, wobei die Seite b Lisas Abstand zur y-Achse und die Seite a der Abstand des Schnittpunktes der beiden Wege S in y-Richtung von Lisa ist. Somit ist der Abstand zwischen Lisa und dem Punkt $\sqrt{a^2+b^2}$. Uns ist egal, wo der Bus sich im Moment befindet, weil er zwar die Zeitdifferenz am Punkt S verändert, aber die Änderung in Abhängigkeit des Winkels bzw. des Verhältnisses $\frac{a}{b}$ bleibt gleich, d.h. die Position des Busses ist für die Ableitung irrelevant.
Somit können wir die Zeitdifferenz Δt in Abhängigkeit des Faktors r in $a = rb$ bei bestimmten Geschwindigkeiten v_B und v_L bestimmen:

$$\Delta t = \frac{r \cdot b}{v_B} - \frac{\sqrt{(r^2+1)} \cdot b}{v_L} \tag{1}$$

Für die Ableitung gilt folglich:

$$\frac{d}{dr}\left(\frac{r \cdot b}{v_B} - \frac{\sqrt{r^2+1} \cdot b}{v_L}\right) = \frac{b}{v_B} + \frac{b \cdot r}{v_L\sqrt{r^2+1}} \tag{2}$$

Nun suchen wir Extremstellen:

$$\frac{d}{dr}\Delta t = 0 \Leftrightarrow r = \pm\frac{v_L}{\sqrt{v_B^2 - v_L^2}} \tag{3}$$

Da wir nur an positiven r interessiert sind, vernachlässigen wir die Vorzeichen an dieser Stelle. Somit wissen wir, wie Lisa laufen muss, wenn ihr kein Hindernis im Weg ist. Den tatsächlichen Winkel auszurechnen ist auf Grund der Modellierung, die ohne Winkel auskommt, unnötig.

1.2 Relevante Punkte

Ich habe bereits erwähnt, dass der Weg bei freier Strecke geradlinig verlaufen muss. Dieser Umstand gilt mehr oder weniger auch, wenn Hindernisse im Weg sind. Denn da Lisa keine Form der Trägheit hat, ist es für sie am effizientesten, stets solange geradeaus zu laufen, bis sie ihre Richtung ändern kann, weil der nächste Wegpunkt in Sicht ist. Das bedeutet, dass sich Lisas Pfad, abgesehen vom Schnittpunkt mit der Straße, ausschließlich aus Eckpunkten der Hindernisse zusammensetzt. Um dies etwas zu präzisieren, kann sogar gesagt werden, dass der Pfad nur aus konvexen Eckpunkten der Polygone besteht. Dies kann man sich leicht erklären, indem man ein konkaves Polygon betrachtet. Zwischen zwei konvexen Ecken verläuft der kürzeste Weg nie über eine konkave Ecke. Sollte nun ein anderes Polygon diesen direkten Weg blockieren, ist auch hier die kürzeste Strecke über die konvexen Ecken jenes Polygons.
Wenn wir nun also versuchen, alle relevanten Punkte zu finden, müssen wir alle konvexen Ecken der Polygone betrachten.

1.3 Kanten

Um herauszufinden, welche Punkte direkt durch eine Kante miteinander verbunden werden können, brauchen wir einen Schnittalgorithmus. Zu der Funktionsweise dieses Algorithmus werde ich stärker in der Umsetzung thematisieren. Relevant ist im Moment, dass er existiert. Wir können also überprüfen, ob zwei Strecken in unserer Ebene sich schneiden. Somit können wir alle diejenigen Wegpunkte miteinander verbinden, zwischen denen die Strecke mit keiner Kante eines Polygons kollidiert.
Die Länge dieser Kante herauszufinden ist nun trivial, weil sie mit dem Satz des Pythagoras und den Koordinaten der beiden Punkte beschrieben werden kann.

1.4 Optimaler Weg

Es gibt viele Algorithmen, um einen optimalen Weg zwischen zwei Punkten in einem Graphen zu finden. Leider bringen uns diese Algorithmen nur etwas, wenn wir die potentiellen Punkte auf der Straße markieren, die erreichbar sind. Darauf möchte ich jedoch verzichten, weil es einen besseren Ansatz gibt. Statt die Punkte zu suchen, die wir auf der Straße optimalerweise erreichen können, schauen wir nur, welche Punkte des Graphen unter dem optimalen Winkel einen freien Weg zur Straße haben und markieren diese.
Beginnend bei dem Startknoten, Lisas Haus, starten wir nun unsere Suche nach dem optimalen Weg. Dabei fügen wir in jedem Schritt einer Warteschlange all jene Pfade hinzu, die aus der aktuellen Instanz

entstehen können. Wir fügen dem bisher gelaufenen Pfad also alle unbesuchten Nachbarn einzeln hinzu. Die Warteschlange sortiert hierbei alle Teilpfade nach der minimalen Zeitdifferenz, für die dieser Pfad sorgen kann.
Diese minimale Differenz ist hierbei eine tatsächliche Untergrenze. Sie berechnet sich aus der Länge der bereits gelaufenen Strecke sowie der Länge des Weges, den Lisa wählen sollte, wenn es ab dem aktuellen Knoten kein Hindernis mehr gäbe. Damit wird klar, dass der optimale Weg genau dann gefunden wurde, wenn der letzte besuchte Knoten des obersten Teilpfades auf der Warteschlange einen freien, optimalen Weg zur Straße hat. Denn in diesem Fall wird die Untergrenze zur tatsächlichen Zeitdifferenz.
Wenn diese Folge von Knoten in unserem Graphen gefunden wurde, wird es mehr oder weniger trivial, den Zeitpunkt zu bestimmen, an dem Lisa spätestens loslaufen muss, um den Bus zu bekommen. Dieser Zeitpunkt ist die Abfahrtzeit des Busses plus die Zeitdifferenz (die aller Wahrscheinlichkeit nach negativ ist).

2 Umsetzung

Ich habe das Programm auf einem Linux-System in der Programmiersprache Go implementiert. Die Wahl der Sprache hat mehrere Gründe. Einerseits ist die Sprache sehr leicht verständlich und ist daher in meinen Augen ähnlich wie Python für den BwInf sehr geeignet und andererseits ist die Sprache schnell. Zum einen lassen sich Projekte sehr schnell kompilieren und zum anderen sind sie ähnlich schnell wie solche, die in Sprachen wie C/C++ geschrieben wurden. Ein weiterer Vorteil, den ich unter Umständen ausnutzen werde, ist die Tatsache, dass Go sehr gute Unterstützung für Nebenläufigkeit bereitstellt. Hinzu kommt der lobenswerte Umstand, dass Go sehr einheitlich ist. Der Quellcode wird automatisch formatiert und die Sprache zwingt mich, meinen Code (in MaSSen) zu kommentieren. All diese Eigenschaften machen Go zu einer der geeignetsten Sprache für die Lösung der Probleme im Rahmen des Bundeswettbewerbs Informatik.
Das entstandene Programm ist eine Applikation, die aus der Command Line einen Server startet, auf den dann im Browser zugegriffen werden kann. Da es für die Lösung der Aufgabe nicht relevant ist, wie ich die Dateien einlese und wie ich meinen Output ausgebe, werde ich diese Schnittstellen nicht weiter thematisieren.
Was hingegen wichtig ist, sind die Datenstrukturen, die genutzt werden, um den Graphen zu erstellen. Hierzu gibt es vier geometrische Objekte, die im Quellcode abstrahiert werden. Diese Objekte befinden sich im `internal`-Paket des Projektes. Das kleinste Element ist ein einfacher Punkt. Dieser besteht aus zwei Gleitkommazahlen, den Koordinaten X und Y.

```
type Point struct {
  X, Y float64
}
```

Hinzu kommen Geraden, die in der *abc*-Form dargestellt werden:

```
type Line struct {
  A, B, C float64
}
```

Es gilt also:

$$ax + by = c \tag{4}$$

Da das Problem nahezu ausschließlich Strecken umfasst, wird auch hierfür eine Datenstruktur benötigt:

```
type LineSegment struct {
  L    *Line
  A, B *Point
}
```

Wie man sieht, setzt sich eine Strecke einerseits aus den beiden Punkten zusammen, die ihr Ende bilden und andererseits aus der Geradengleichung, die den Verlauf der Strecke beschreibt.
Wenn man nun Polygone beschreiben will, kann man sich entscheiden, ob diese als geordnete Menge von Punkten oder von Strecken beschrieben werden. Ich habe mich für Punkte entschieden, weil die Inputdaten

ebenfalls in diesem Format vorhanden sind und Strecken im Wesentlichen auch nur Paare von Punkten sind. Um die späteren Berechnungen einfacher zu machen, nehmen wir eine weitere Abstraktion vor. Die Ecken der Polygone sind nun nicht nur einfache Punkte, sondern kennen weiterhin ihre Nachbarn sowie das Polygon dem sie angehören.

```
type Corner struct {
  *Point
  Plygn *Polygon
  N1    *Corner
  N2    *Corner
}
```

Damit lässt sich ein Polygon folgendermaßen beschreiben:

```
type Polygon struct {
  Corners []*Corner
  N       int
}
```

Dabei ist `Polygon.N` die Anzahl der Ecken des Polygons.
Nun lesen wir die Eingabedaten und konstruieren das Gelände, in dem Lisa sich bewegen muss. Dieses Gelände wird als `internal/terrain.Terrain` abstrahiert.

```
type Terrain struct {
  PolCnt int
  Plygns []*internal.Polygon
  Start  *internal.Point
}
```

Dabei ist `Start` der Punkt, an dem Lisa zu laufen beginnt. Um aus diesem Gelände jetzt einen Graphen zu generieren, benötigen wir folgende Schritte.
Zunächst müssen wir die Polygone einlesen. Der Vorgang an sich ist einfach, interessant ist jedoch, was man bereits in diesem Schritt machen kann, um spätere Untersuchungen einfacher zu gestalten. Beispielsweise werden hierbei alle Polygone so angeordnet, dass sie im Uhrzeigersinn abgelaufen werden. Dies geschieht mit folgender Funktion:

```
func (P *Polygon) isClockwise() bool {
  var s float64
  for _, c := range P.Corners {
    if c.N2.Point.Y != -c.Point.Y {
      s += (c.N2.Point.X - c.Point.X) /
        (c.N2.Point.Y + c.Point.Y)
    }
  }
  return s > 0
}
```

Diese habe ich bei meiner Recherche auf StackOverflow gefunden: `https://stackoverflow.com/questions/1165647/how-to-determine-if-a-list-of-polygon-points-are-in-clockwise-order`. Ein weiterer interessanter Trick ist es, alle Ecken wegzulassen, an denen der Winkel 180 Grad beträgt. Das hängt damit zusammen, dass diese Ecken nichts zum optimalen Weg beiträgt, genauso wenig wie konkave Ecken. Im Gegensatz zu diesen ändert es aber auch nichts am optimalen Weg, diese Ecken einfach wegzulassen. Dies geschieht mit dieser Funktion:

```
func (P *Polygon) removeFlatCorners() {
  removed := 0
  for i, c := range P.Corners {
    ls := NewLineSegment(*c.N1.Point, *c.N2.Point)
    if ls.L.RelationOf(c.Point) == 0 {
      P.remove(i - removed)
      removed++
    }
  }
}
```

Es wird also überprüft, ob eine Ecke auf der Strecke liegt, die ihre beiden Nachbarn im Polygon verbindet. Anschließend müssen wir nacheinander alle Ecken der Polygone ablaufen und entscheiden, ob der Punkt relevant ist. Wenn er das ist, wird er der Liste der relevanten Punkte hinzugefügt.

```
pt2indx := make(map[*internal.Point]int)
indx2pt := make(map[int]*internal.Point)
pt2indx[t.Start] = 0
indx2pt[0] = t.Start
count := 1
for _, plygn := range t.Plygns {
  for _, c := range plygn.Corners {
    if !c.IsConcave() {
      indx2pt[count] = c.Point
      pt2indx[c.Point] = count
      count++
    }
  }
}
```

Wie man sieht, erstellen wir zwei Maps (im Wesentlichen Hashtabellen), die einerseits Indizes auf Punkte und andererseits Punkte auf Indizes abbilden. Diesen Maps fügen wir nun zunächst Lisas Zuhause und anschließend alle konvexen Ecken der Polygone hinzu.
Wie finden wir heraus, ob eine Ecke konvex ist? Ganz einfach:

```
func (crnr *Corner) IsConcave() bool {
  a := crnr.N1.Point
  b := crnr.Point
  c := crnr.N2.Point
  return (b.X-a.X)*(c.Y-b.Y)-(b.Y-a.Y)*(c.X-b.X) < 0
}
```

Dieser Zusammenhang entstammt diesem StackOverflow-Thread: `https://stackoverflow.com/questions/471962/how-do-i-efficiently-determine-if-a-polygon-is-convex-non-convex-or-complex/`. Im Wesentlichen handelt es sich hierbei um die Richtung des Vektorproduktes, welches angibt, in welche Richtung der Winkel verläuft. Da die Ecken aller Polygone eingangs so sortiert wurden, dass jedes Polygon im Uhrzeigersinn abgelaufen wird, bleibt die nötige Richtung für konvexe Winkel stets die gleiche.
Nun müssen wir für all diese relevanten Punkte bestimmten, welche paarweisen Verbindungen bestehen.

```
mtrx := make([]float64, count*count)
var d float64
for i := 0; i < count; i++ {
  for j := 0; j < i; j++ {
    p1 := indx2pt[i]
    p2 := indx2pt[j]
    if !HasIntersections(t, internal.NewLineSegment(*p1, *p2)) {
      d = math.Sqrt(internal.GetSqDist(p1, p2))
    } else {
      d = -1
    }
    mtrx[i*count+j] = d
    mtrx[j*count+i] = d
  }
}
```

Offensichtlich können wir eine Kante nur dann in den Graphen einfügen, wenn die gesamte Strecke keine Schnitte mit irgendeinem anderen Polygon aufweist. Wenn solch eine Verbindung besteht, kann der Abstand der beiden Punkte mit Hilfe des Satzes des Pythagoras bestimmt und als Kantengewicht festgeschrieben werden. Nun müssen wir nur noch herausfinden, wann eine Strecke einen Schnittpunkt mit einem Polygon der Ebene hat.
Dazu überprüfen wir für jedes Polygon, ob diese Strecke jenes Polygon schneidet. Um das zu tun, berechnen wir für jede Ecke des jeweiligen Polygons, ob sie über, unter oder auf der Geraden liegt, die unsere Strecke enthält. Dabei ist bei vertikalen Geraden links oben und rechts unten.
Mit diesen Informationen müssen wir nun zunächst prüfen, ob es einen Schnitt gibt, bei dem eine Ecke auf der Geraden liegt. Solch eine Situation festzustellen, geht wie folgt: Wenn der Punkt auf der Geraden

liegt, muss von dem Eckpunt aus für die beiden benachbarten Ecken und die beiden Endpunkte der Geraden der absolute Winkel im Koordinatensystem bestimmt werden. Diesen kann man sich beispielsweise als Winkel zwischen der jeweiligen Strecke und der x-Achse vorstellen. Nun dürfen die Winkel, wenn sie der Größe nach sortiert sind, nicht alternieren. D.h. die beiden Endpunkte dürfen nicht durch die Winkel der benachbarten Ecken separiert werden. Sollte dies doch so sein, so existiert ein Schnitt durch diese Ecke.
Der Code hierfür sieht wesentlich komplizierter aus, als er eigentlich ist:

```
func (LS *LineSegment) inCounterClockwiseAngle(p0, p1, p2 *Point) bool {
  angle1, err := GetAngle(p0, p1)
  if err != nil {
    return false
  }
  angle2, err := GetAngle(p0, p2)
  if err != nil {
    return false
  }
  angleA, errA := GetAngle(p0, LS.A)
  angleB, errB := GetAngle(p0, LS.B)
  condA := angleA > angle1 && angleA < angle2
  condB := angleB > angle1 && angleB < angle2
  if errA == nil && errB == nil {
    return condA != condB
  }
  if errA == nil {
    return condA
  } else if errB == nil {
    return condB
  }
  return false
}
```

Sollte es keinen derartigen Schnittpunkt geben, muss nach klassischen Schnitten zwischen zwei Strecken gesucht werden. Da wir jedoch schon wissen, wie sich die einzelnen Punkte zueinander verhalten, ist auch das sehr einfach. Sollten die Relationen zweier benachbarter Punkte im Polygon so zur gegebenen Strecke sein, dass sie auf verschiedenen Seiten liegen, muss nur noch die umgedrehte Abfrage durchgeführt werden. D.h. nun wird überprüft, ob die beiden Enden der gegebenen Strecke auf verschiedenen Seiten der Polygonseite liegen. Wenn beide Bedingungen wahr sind, muss es einen Schnittpunkt geben. Der Vorteil dieser Herangehensweise ist, dass keine Rechenleistung dafür verschwendet wird, den tatsächlichen Punkt auszurechnen, da wir offensichtlich nicht daran interessiert sind, wo dieser Punkt liegt.
Nun haben wir einen Graphen, der mit allen nötigen Kanten und Längen ausgestattet ist. Für die einzelnen Beispieldatein sieht das folgendermaßen aus.

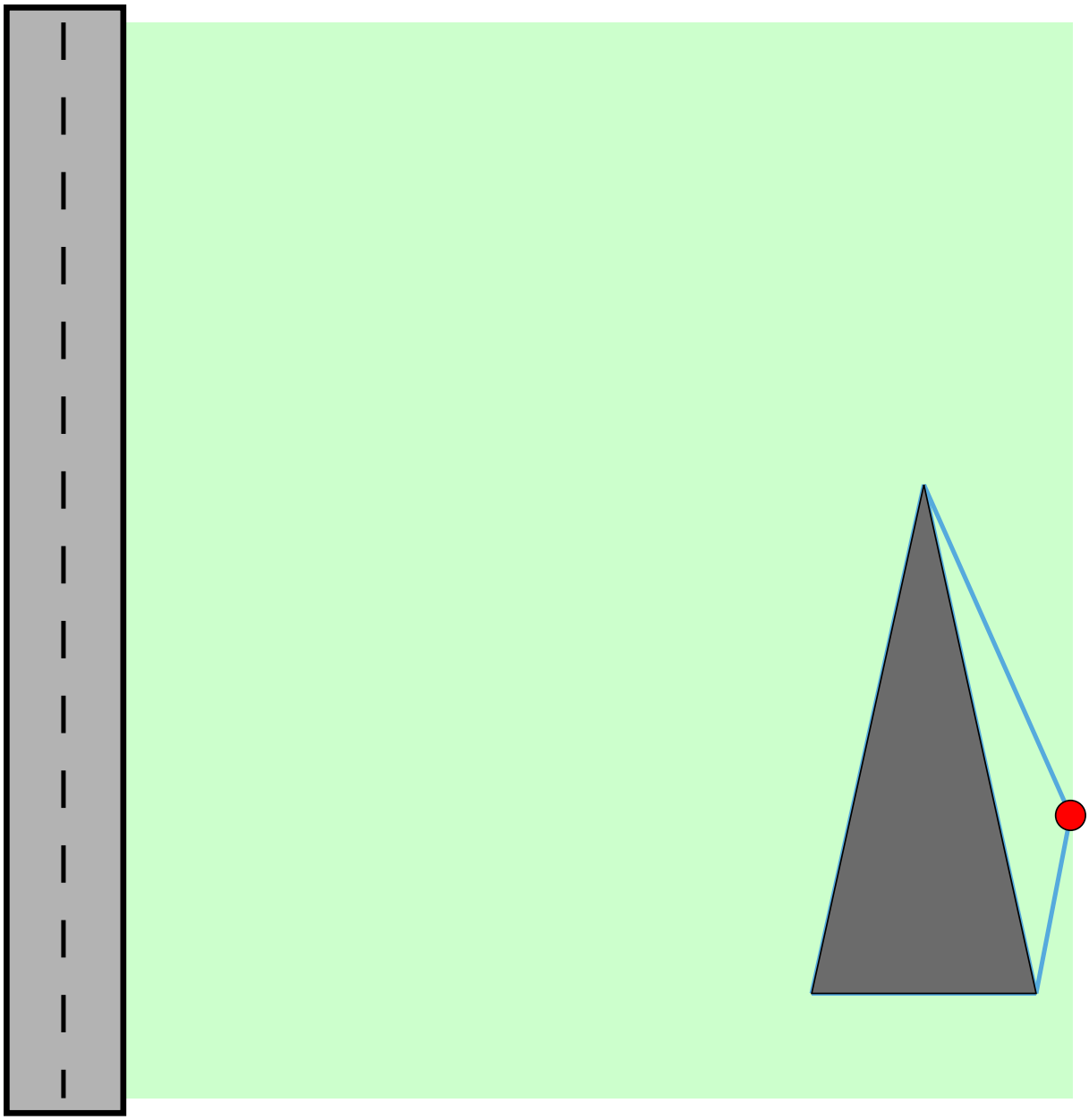

Abbildung 1: Beispiel 1 Graph

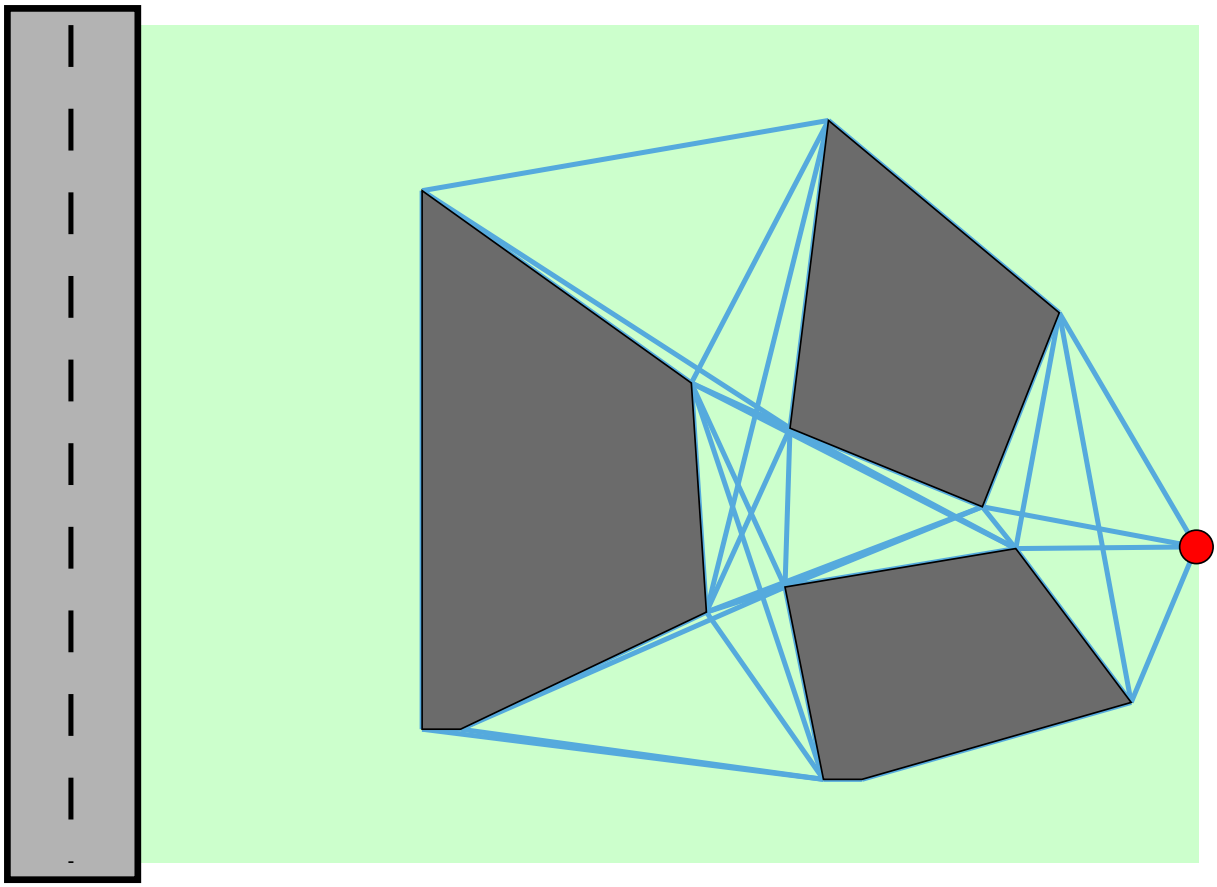

Abbildung 2: Beispiel 2 Graph

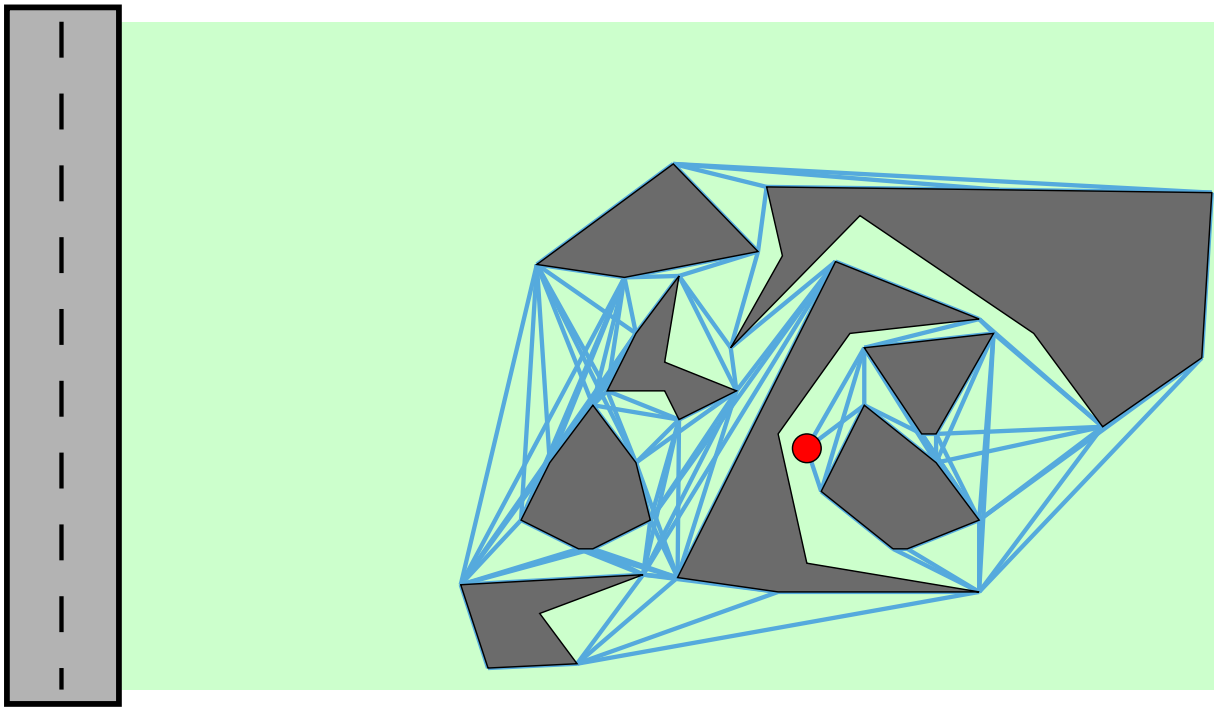

Abbildung 3: Beispiel 3 Graph

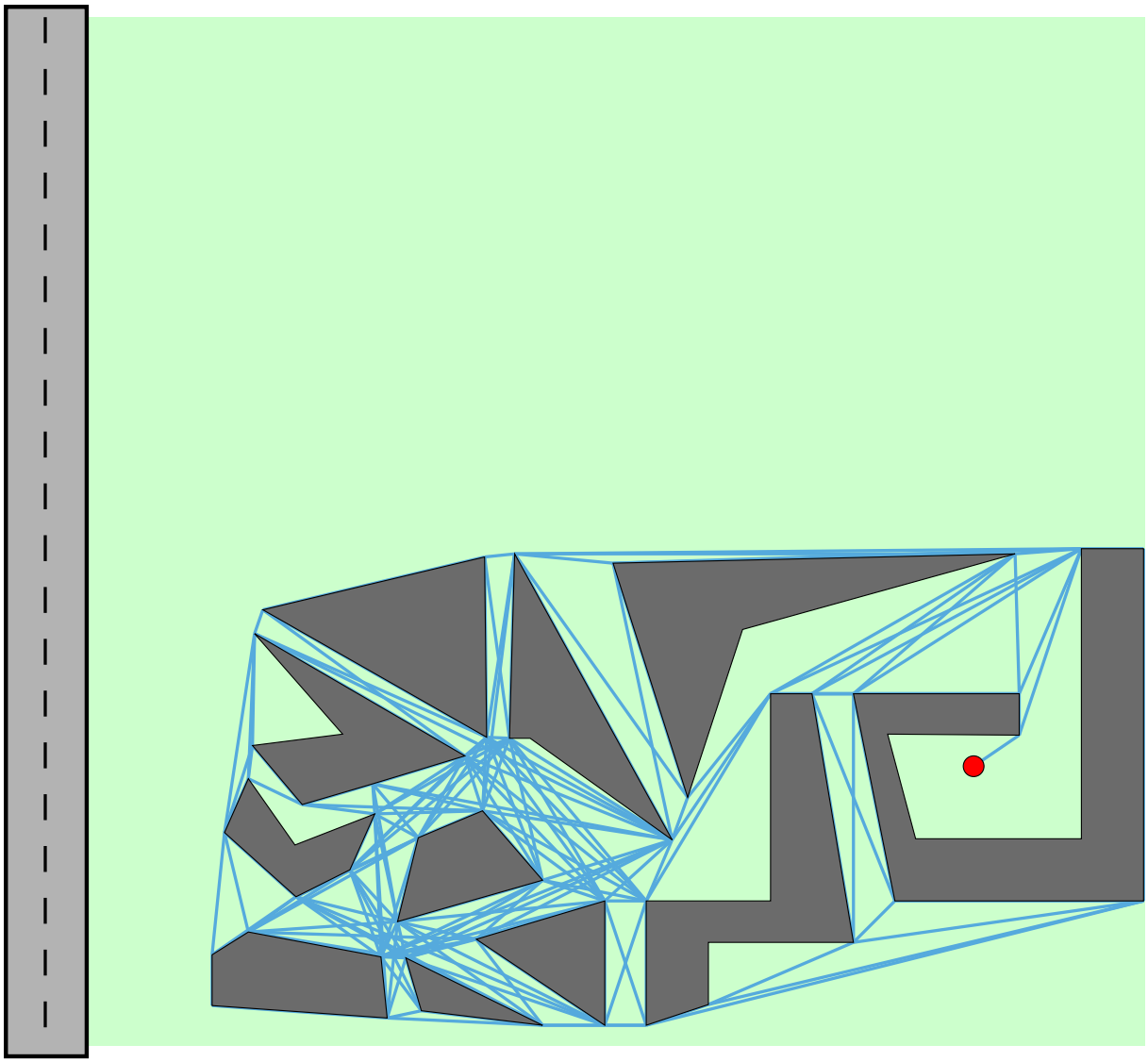

Abbildung 4: Beispiel 4 Graph

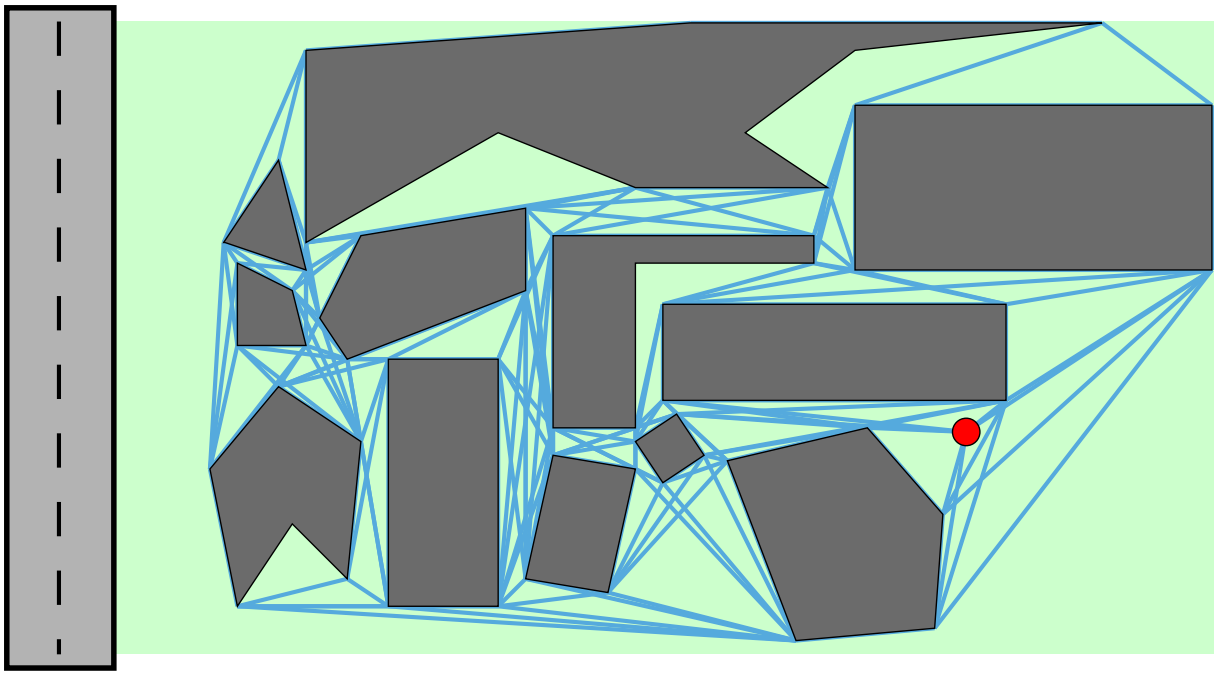

Abbildung 5: Beispiel 5 Graph

Da ich diese SVG-Dateien bereits generiert hatte, bevor das neue Template zur Verfügung stand, sehen sie jetzt mehr aus wie das Alte. Ich bitte um Verzeihung. Nun müssen wir in diesem Graphen noch den kürzesten Pfad zur Straße finden. Den Algorithmus dafür habe ich bereits in Kapitel 1 erklärt.

```
bck := make([]int, g.N)
for i := range bck {
  bck[i] = -1
}
start := &partPath{bck, 0, g.maxDelta(g.indx2pt[0], 0), 0}
q := &prioQ{make([]*partPath, segLen), 0, g.getFinals()}
q.Put(start)
var i int
var part *partPath
for part = q.Get(); !q.final[part.last]; part = q.Get() {
  for i = 0; i < g.N; i++ {
    if part.bck[i] == -1 && g.Mtrx[part.last*g.N+i] >= 0 {
      q.Put(g.extend(part, i))
    }
  }
}
```

Interessant ist hier wahrscheinlich nur noch die Prioritätswarteschlange. Diese enthält Pointer zu den Teilpfaden in der Schlange, eine Länge und ein boolsches Array, welches für all die Knotenindizes den Wert `wahr` enthält, die einen direkten Weg zur Straße in optimaler Richtung haben.

```
type prioQ struct {
  data  []*partPath
  len   int
  final []bool
}
```

Die zwei essenziellen Operationen sind `Put` und `Get`.

```
func (q *prioQ) Put(e *partPath) {
  q.len++
  if q.len%segLen == 0 {
    newData := make([]*partPath, q.len+segLen)
    copy(newData, q.data)
    q.data = newData
  }
```

```
    q.data[q.len-1] = e
    q.up(q.len - 1)
}

func (q *prioQ) Get() *partPath {
    if q.len == 0 {
        return nil
    }
    v := q.data[0]
    q.len--
    q.data[0] = q.data[q.len]
    q.down(0)
    return v
}
```

Wie man sieht handelt es sich bei der Implementation um einen binären Heap. Diese Datenstrukturen skalieren zwar nicht so gut wie andere Heaps, verbrauchen jedoch weniger Arbeitsspeicher und sind praktisch sehr schnell. Jeder der schonmal einen solchen Heap gesehen hat, sollte jetzt von der Implementation nicht sonderlich überrascht sein. Aus diesem Grund verzichte ich darauf, die beiden wesentlichen Funktionen zur Erhaltung der Heapstruktur, `up` und `down`, zu erläutern. Wichtig zu wissen ist, dass der Vergleichswert der PQ das `max`-Attribut der Elemente ist. Je kleiner, desto höher ist die Priorität.
Nun enthält jeder Teilpfad ein Array in dem an der Stelle jedes besuchten Knotens der Index des Vorgängerknotens und an allen anderen Stellen -1 steht. Des Weiteren ist ein Wert vorhanden, der angibt, wie lang der Weg bisher schon ist, und einer, der angibt, wie lang der Weg maximal werden kann. Zuletzt gibt es noch eine Angabe, welcher Knoten als letztes besucht wurde, was sehr viel Zeit spart, weil somit sehr schnell die Pfade erweitert werden können.

3 Ergebnisse

Das Program heißt `lisa-rennt` und ist unter Linux auführbar. Es akzeptiert den Pfad zu der einzlesenen Datei als Argument in der Kommandozeile. Durch das Ausführen des Programmes wird ein HTTP-Server auf `127.0.0.1:8080` gestartet. Sobald man eine HTTP-Anfrage an den Server schickt, werden alle wichtigen Informationen in der Konsole ausgegeben. Des Weiteren sollte auf der geladenen Seite die SVG Datei zu sehen sein. Auf diese kann man auch unter `<IP>/data/[graph/terrain/path/all]` direkt zugreifen. Sollte das Wechseln der Anzeige nicht direkt funktionieren, kann es helfen mehrmals den dafür vorgesehenen Button zu drücken.
Die entstandenen Graphen habe ich bereits oben abegebildet. Nun folgen die optimalen Pfade für die Beispiele sowie all die Angaben, die gefordert sind.

3.1 Beispiel 1

```
Lisa...
...muss 7:29:39 Uhr losgehen...
...wird den Bus 7:30:37 Uhr treffen.
...wird den Bus 308.88 Meter von der Haltestelle entfernt treffen.
...muss 241.75 Meter laufen und benötigt 58.02 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 633.00, y: 189.00]
Polygon: P1, Punkt: [x: 535.00, y: 410.00]
Polygon: S, Punkt: [x: 0.00, y: 308.88]
```

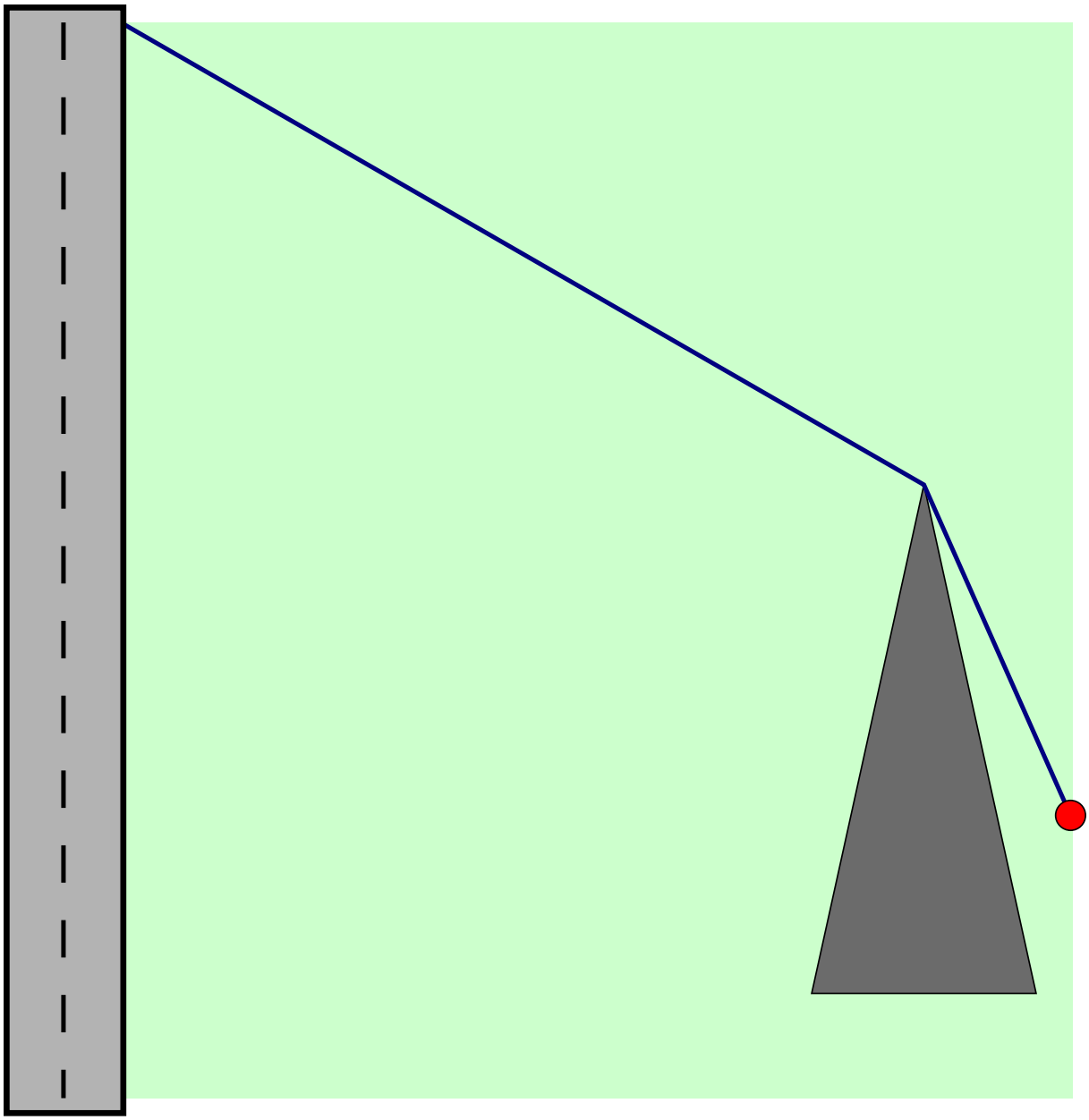

Abbildung 6: Beispiel 1 Pfad

3.2 Beispiel 2

```
Lisa...
...muss 7:28:7 Uhr losgehen...
...wird den Bus 7:30:11 Uhr treffen.
...wird den Bus 98.15 Meter von der Haltestelle entfernt treffen.
...muss 516.31 Meter laufen und benötigt 123.91 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 633.00, y: 189.00]
Polygon: P1, Punkt: [x: 505.00, y: 213.00]
Polygon: P1, Punkt: [x: 390.00, y: 260.00]
Polygon: P3, Punkt: [x: 170.00, y: 402.00]
Polygon: S, Punkt: [x: 0.00, y: 98.15]
```

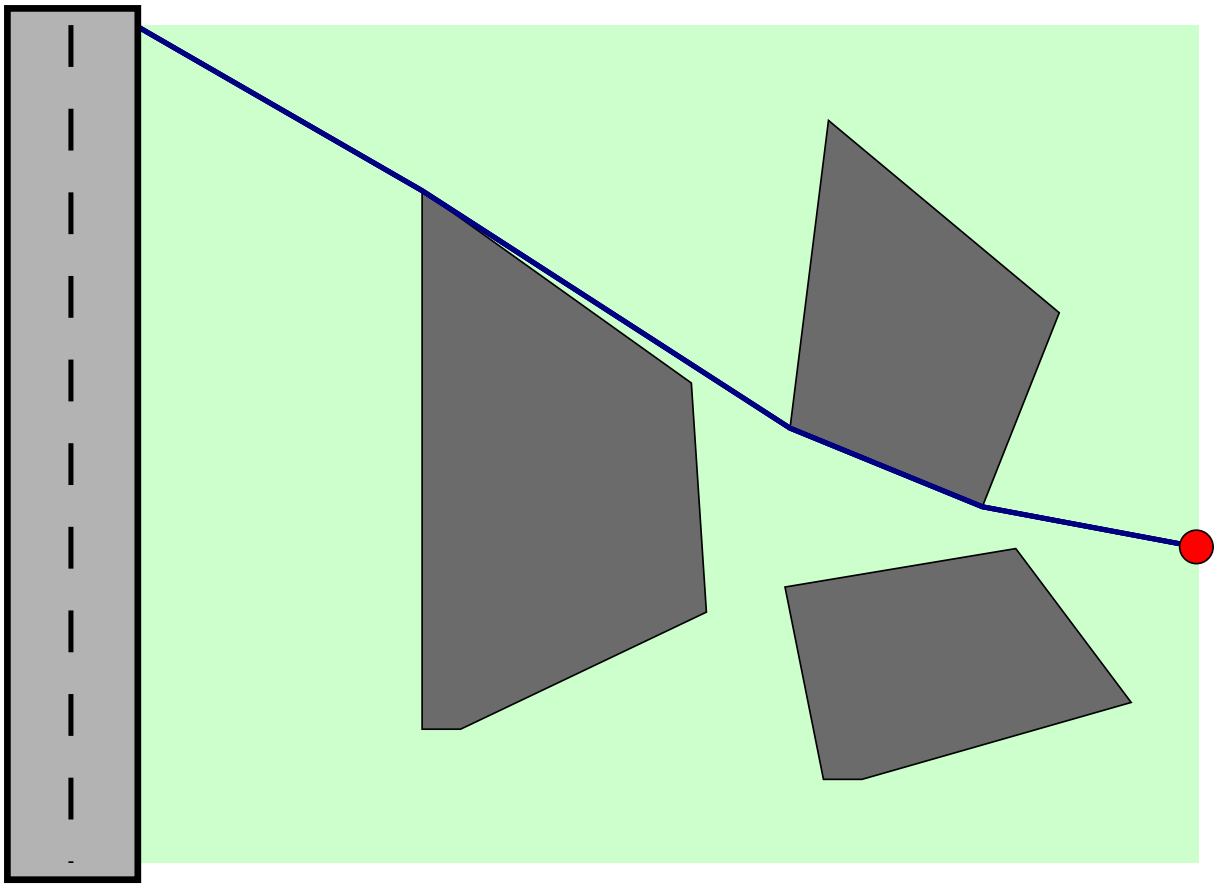

Abbildung 7: Beispiel 2 Pfad

3.3 Beispiel 3

```
Lisa...
...muss 7:28:13 Uhr losgehen...
...wird den Bus 7:30:20 Uhr treffen.
...wird den Bus 168.01 Meter von der Haltestelle entfernt treffen.
...muss 526.57 Meter laufen und benötigt 126.38 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 479.00, y: 168.00]
Polygon: P2, Punkt: [x: 519.00, y: 238.00]
Polygon: P3, Punkt: [x: 599.00, y: 258.00]
Polygon: P3, Punkt: [x: 499.00, y: 298.00]
Polygon: P8, Punkt: [x: 426.00, y: 238.00]
Polygon: P5, Punkt: [x: 390.00, y: 288.00]
Polygon: P6, Punkt: [x: 352.00, y: 287.00]
Polygon: P6, Punkt: [x: 291.00, y: 296.00]
Polygon: S, Punkt: [x: 0.00, y: 168.01]
```

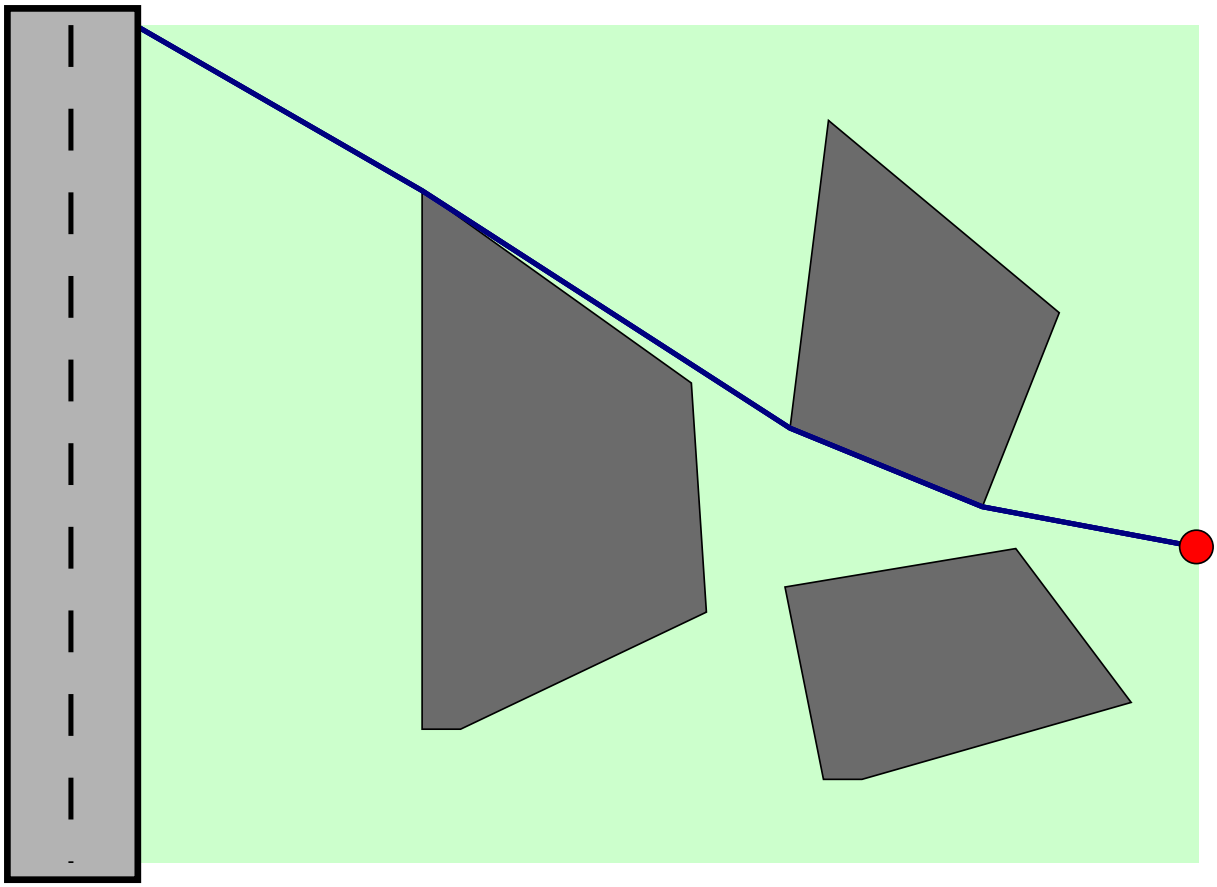

Abbildung 8: Beispiel 3 Pfad

3.4 Beispiel 4

```
Lisa...
...muss 7:30:7 Uhr losgehen...
...wird den Bus 7:31:2 Uhr treffen.
...wird den Bus 517.31 Meter von der Haltestelle entfernt treffen.
...muss 228.31 Meter laufen und benötigt 54.80 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 856.00, y: 270.00]
Polygon: P11, Punkt: [x: 900.00, y: 300.00]
Polygon: P11, Punkt: [x: 900.00, y: 340.00]
Polygon: P10, Punkt: [x: 896.00, y: 475.00]
Polygon: S, Punkt: [x: 0.00, y: 517.31]
```

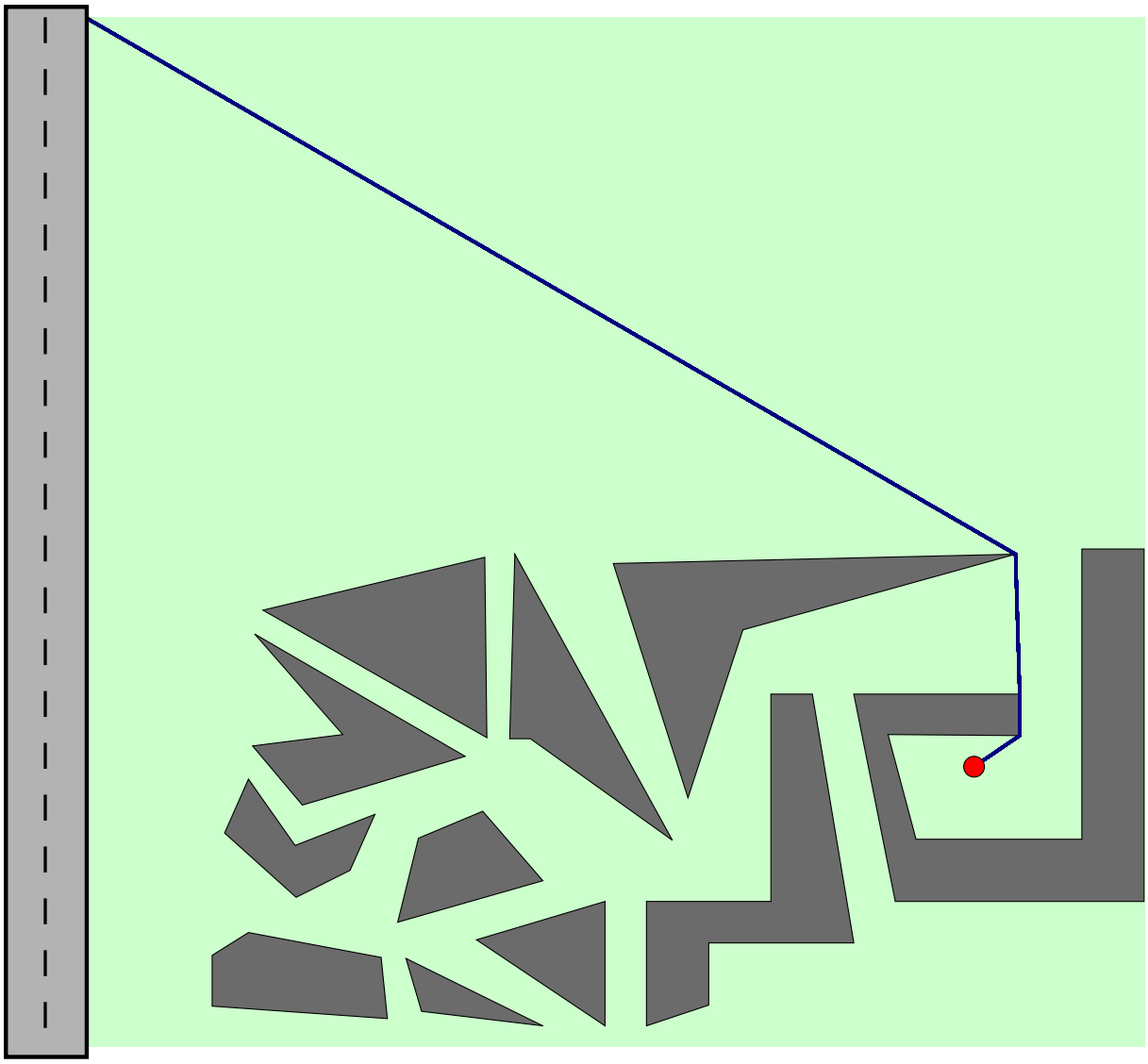

Abbildung 9: Beispiel 4 Pfad

3.5 Beispiel 5

```
Lisa...
...muss 7:30:7 Uhr losgehen...
...wird den Bus 7:31:2 Uhr treffen.
...wird den Bus 517.31 Meter von der Haltestelle entfernt treffen.
...muss 228.31 Meter laufen und benötigt 54.80 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 856.00, y: 270.00]
Polygon: P11, Punkt: [x: 900.00, y: 300.00]
Polygon: P11, Punkt: [x: 900.00, y: 340.00]
Polygon: P10, Punkt: [x: 896.00, y: 475.00]
Polygon: S, Punkt: [x: 0.00, y: 517.31]
```

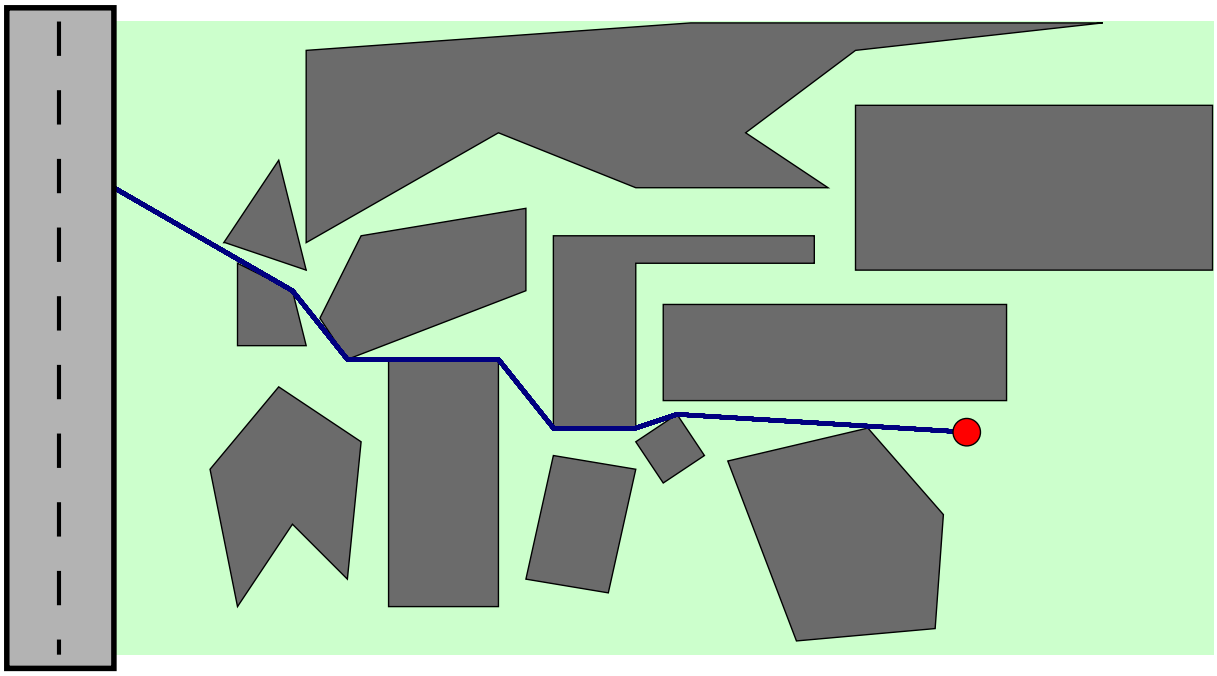

Abbildung 10: Beispiel 5 Pfad

4 Erweiterung

Da mir die Beispieldaten sehr klein vorkamen und ich einen Weg suchte, herauszufinden, was mein Programm aushält, musste ich mir Gedanken darüber machen, wie ich an bessere Eingabedaten kommen könnte. Meine erste Idee waren Höhendaten, aus denen man die Höhenlinien extrahieren könnte. Als ich da nicht fündig wurde, hatte ich die Idee, Bilder mit einem Filter in ausschlieSSlich schwarze und weiSSe Pixel zu zerlegen und die entstandenen Flächen zu vektorisieren. Dabei sollte man nicht jeden Pixel am Rand einer Fläche als Eckpunkt wählen, weil sonst nur acht verschiedene Anstiege der Kanten möglich sind. Wenn man jedoch beispielsweise nur jeden fünften Pixel wählt, wird die Vielfalt an Anstiegen ausreichend groß.
Nun stellt sich die Frage: Welche Bilder kann man mit welchem Filter geeignet zerlegen? Die beste Antwort, die ich darauf finden konnte waren Bilder von Text. Denn hier muss nichtmal etwas gefiltert werden. Text ist im wesentlichen nur eine Menge schwarzer Polygone auf weiSSem Grund. Also habe ich mir Text gewählt und diesen in ein Bild konvertiert. (`https://convertio.co/de/txt-jpg/`)
Nun musste der Text noch in passende Eingabedaten zerlegt werden. Hierzu habe ich ein weiteres Programm geschrieben. Das Programm und der Quellcode sind im Ordner `programme/vectorize` zu finden. Das Programm filtert ein Bild so, dass jeder Pixel, bei dem die Summe der Farbwerte (R, G und B) größer als 600 ist, als weiß und jeder andere Pixel als schwarz markiert wird. Nun wird der erste schwarze Pixel von oben links nach unten rechts gesucht. An dieser Stelle wird ein weißer Nachbar und im Anschluss der nächste Schwarze nachbar im Uhrzeigersinn gesucht. Dabei beginnt die Suche nach weiSSen Nachbarn bei allen weiteren Pixeln immer beim Vorgänger, d.h. einem schwarzen Pixel, und fährt dann ebenfalls im Uhrzeigersinn fort. Dadurch sollte der Startpixel immer erst dann wiedergefunden werden, wenn der Rand des gesamten Polygons abgelaufen wurde. Da mir egal ist, ob die Polygone nun genau so aussehen, wie der Text, den ich als Eingabe gewählt habe, musste ich dieses Verfahren nicht weiter optimieren.
Das Ergebnis sind nun groSSe Mengen an Polygonen, die in die Standardausgabe geschrieben werden und die genutzt werden können, um das eigentliche Program zu testen. Sollten Sie selbst Daten generieren wollen, sollten Sie beachten, dass noch die erste Zeile, also die Anzahl der Polygone, und die letzte Zeile, also die Koordinaten von Lisas Haus, eingefügt werden müssen.
Alle Eingaben wurden aus dem beiliegenden Bild `erweiterung.jpg` generiert.

4.1 Eingabe 1

Die erste Eingabedatei ist eine stark gekürzte Version der eigentlichen Ausgabe. Sie enthält 343 Polygone und ist als `erweiterung1.txt` zu finden. Lisas Haus befindet sich mitten in den Polygonen bei

$L(1000, 300)$.

```
Lisa...
...muss 7:29:19 Uhr losgehen...
...wird den Bus 7:30:46 Uhr treffen.
...wird den Bus 389.13 Meter von der Haltestelle entfernt treffen.
...muss 362.10 Meter laufen und benötigt 86.90 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 1000.00, y: 300.00]
Polygon: P155, Punkt: [x: 963.00, y: 311.00]
Polygon: P165, Punkt: [x: 936.00, y: 323.00]
Polygon: P181, Punkt: [x: 878.00, y: 344.00]
Polygon: P249, Punkt: [x: 703.00, y: 438.00]
Polygon: P265, Punkt: [x: 674.00, y: 455.00]
Polygon: S, Punkt: [x: 0.00, y: 389.13]
```

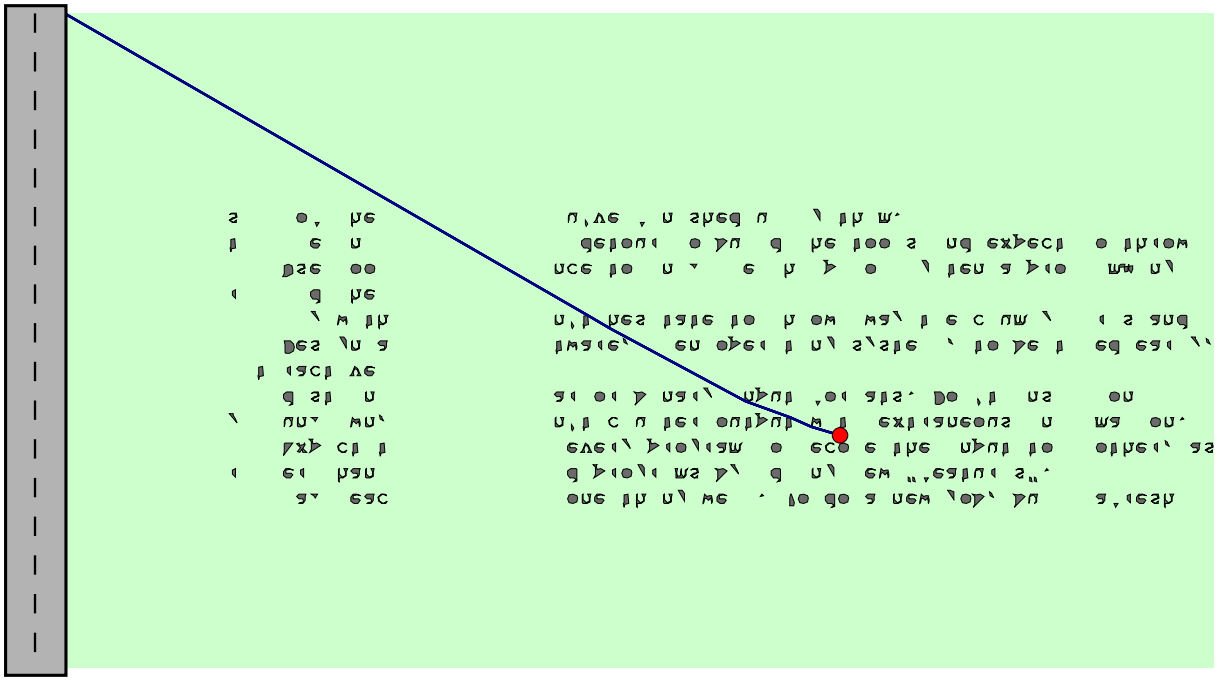

Abbildung 11: Erweiterung 1 Pfad

4.2 Eingabe 2

Die zweite Eingabe enthält alle Polygone, die aus dem Bild generiert wurden. Das sind 529 an der Zahl. Lisas Haus befindet sich jedoch auf der x-Achse und somit etwas entfernt von den Hindernissen.

```
Lisa...
...muss 7:27:1 Uhr losgehen...
...wird den Bus 7:30:19 Uhr treffen.
...wird den Bus 159.35 Meter von der Haltestelle entfernt treffen.
...muss 821.49 Meter laufen und benötigt 197.16 Sekunden.
...muss folgende Wegpunkte passieren:

Polygon: L, Punkt: [x: 1000.00, y: 0.00]
Polygon: P345, Punkt: [x: 604.00, y: 207.00]
Polygon: P357, Punkt: [x: 576.00, y: 224.00]
Polygon: P377, Punkt: [x: 531.00, y: 245.00]
Polygon: P442, Punkt: [x: 398.00, y: 323.00]
```

```
Polygon: P464, Punkt: [x: 360.00, y: 345.00]
Polygon: P474, Punkt: [x: 341.00, y: 356.00]
Polygon: P501, Punkt: [x: 290.00, y: 378.00]
Polygon: P505, Punkt: [x: 276.00, y: 387.00]
Polygon: S, Punkt: [x: 0.00, y: 159.35]
```

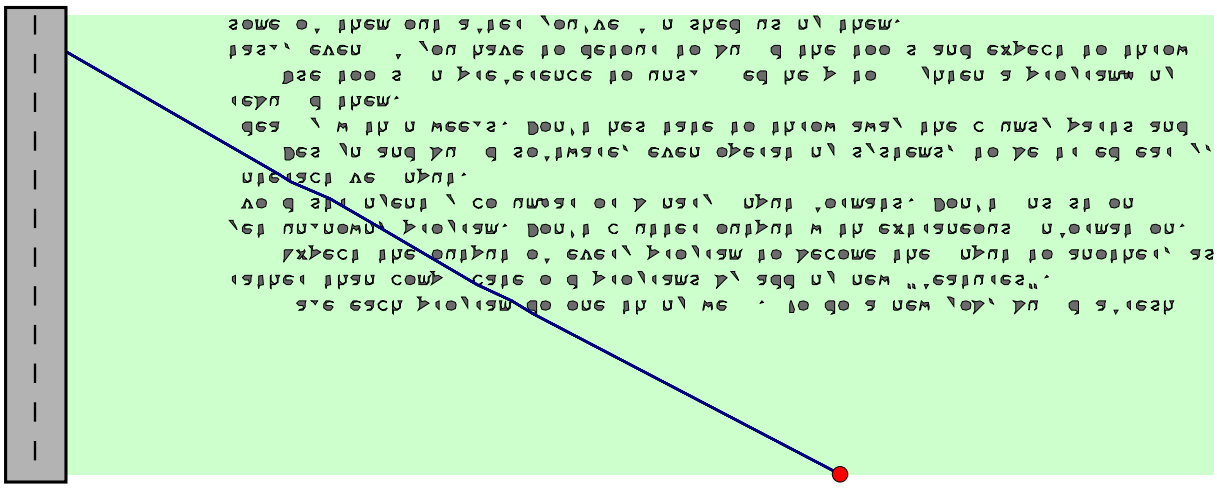

Abbildung 12: Erweiterung 2 Pfad

4.2.1 Eingabe 3

Die dritte Eingabedatei enthält die gleiche Polygone wie die zweite. Der einzige Unterschied ist die Position von Lisas Haus. Dieses wurde nun 100 Meter nach oben bewegt. Dieser kleine Unterschied sorgt jedoch dafür, dass das Program keine optimale Lösung mehr finden kann, weil der Arbeitsspeicher überläuft. Das hängt damit zusammen, dass der Algorithmus zum Finden des Weges nun wesentlich mehr Hindernisse einkalkulieren muss, weil kein ansatzweise direkter Weg existiert.
Um dieses Problem zu lösen wäre es eventuell sinnvoll gewesen mit dem Dijkstra-Algorithmus den kürzesten Weg zu jedem Punkt mit freiem, optimalem Zugang zur StraSSe zu berechnen und anschlieSSend die Kandidaten anhand der Zeitdifferenz am Schnittpunkt mit der StraSSe zu unterscheiden. Diese herangehensweise würde jedoch aller Wahrscheinlichkeit nach für einen Verlust im Bereich der Laufzeit bei den jenigen Eingaben sorgen, die im vorhandenen Arbeitsspeicher lösbar sind.

5 Quellen

Da ich nicht weiß, wie Sie mit Inspirationen u.Ä. umgehen, möchte ich an dieser Stelle etwas zu den Quellen dieser Arbeit sagen. Einerseits habe ich natürlich für viele Sachen StackOverflow verwendet. Die wirklich relevanten Sachen sind jedoch im Quellcode oder hier in der Dokumentation benannt. Eine weitere wichtige Quelle, wenn man das so nennen darf, ist meine eigene Implementation eines Programmes zur Lösung des TSP. (`https://github.com/miltfra/tsp`) Daher stammt der großteil des A*, der hier verwendet wurde, um den kürzesten Pfad im Graphen zu finden.