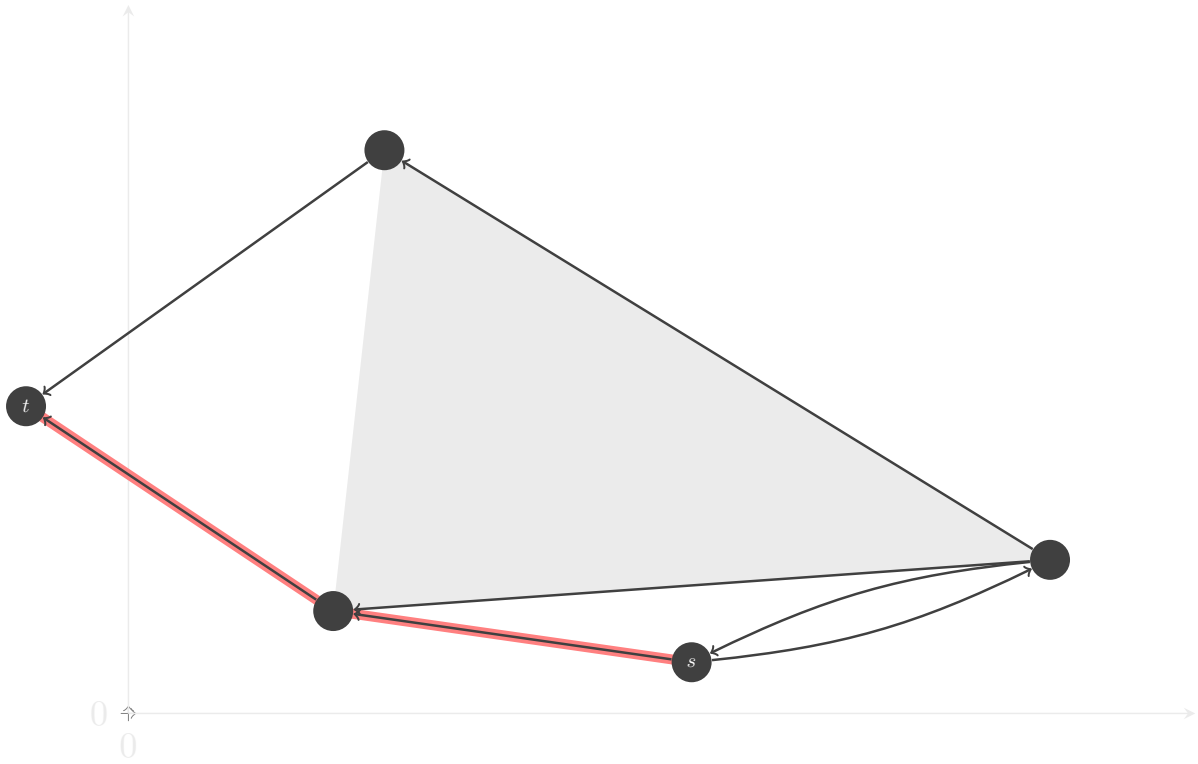
Lisa rennt

Aufgabe 1, Runde 2, 37. Bundeswettbewerb Informatik

Erik Klein

Teilnahme-ID: 52207



Inhaltsverzeichnis

1 Lösungsidee

1.1 Grundlegende Überlegungen

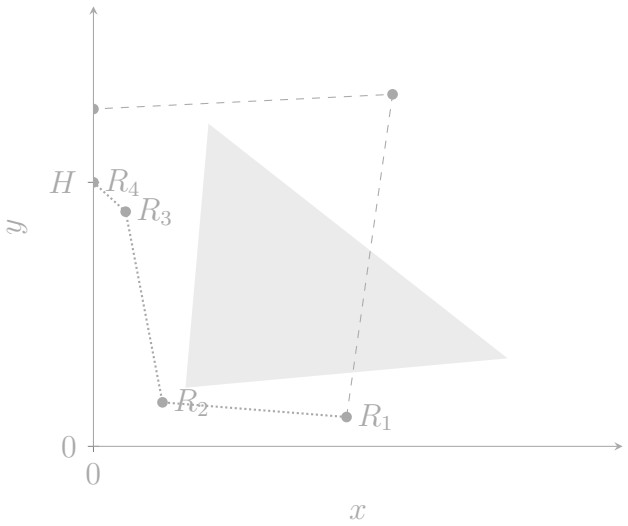
Ich habe folgende Annahmen getroffen:

- Der Bus und Lisa sind punktförmig, der Einfachheit halber.
- Lisa kann sich nur auf Punkten mit $x, y \geq 0$ befinden, worauf hindeutet, dass in der Grafik in der Aufgabenstellung nur ein Wertebereich mit $x, y \geq 0$ angezeigt ist und der Bus nur Punkte mit $y \geq 0$ erreicht. Diese Annahme hat Überlegungen zur optimalen Route erleichtert.

Lisa muss sich vom Startpunkt P_{start} entlang einer Route zu einem Zielpunkt $(\,0\,|\,H\,)$ bewegen. Eine *Route* $R = \langle\, R_1,\, R_2,\, \ldots,\, R_L \rangle$ ist dabei eine Folge von $L \geq 2$ Punkten, den sog. *Eckpunkten*. Sie beginnt mit $R_1 = P_{start}$ und endet mit $R_L = (\,0\,|\,H\,)$.

Da der kürzeste Weg zwischen zwei Punkten die Strecke zwischen ihnen ist, bewegt sich Lisa von einem Eckpunkt zum darauf folgenden jeweils auf der Strecke zwischen ihnen. Keine der Strecken darf ein Hindernis schneiden. Eine Strecke schneidet ein Hindernis, wenn sie einen Bereich innerhalb eines einem Hindernis entsprechenden Polygons, eines sog. *Hindernispolygons*, aufweist.

Somit zeigt in der folgenden Abbildung die punktierte Linie eine Route, die möglich ist. Die gestrichelte Linie zeigt eine Route, die nicht möglich ist, da eine ihrer Strecken ein Hindernis schneidet:



Lisas Startzeitpunkt für eine Route, T_{Lisa}, ist abhängig von:

- der Zeit ΔT_{Bus}, die der Bus benötigt, um sich von $(\,0\,|\,0\,)$ zu $(\,0\,|\,H\,)$ zu bewegen. Da er dafür mit Geschwindigeit 30km/h einen Weg der Länge H zurücklegen muss, gilt:

$$\Delta T_{\text{Bus}} = \frac{H}{30\text{km/h}}$$

- der Zeit ΔT_{Lisa}, die Lisa benötigt, um sich entlang der Route zu bewegen. Diese ist die Summe der Zeiten, die sie für die einzelnen Strecken benötigt.

  Die für eine Strecke benötigte Zeit erhält man, indem man ihre Länge, also den euklidischen Abstand ihrer Endpunkte, durch Lisas Geschwindigkeit, 15km/h, teilt. Mathematisch ausgedrückt:

$$\Delta T_{\text{Lisa}} = \sum_{i=1}^{L-1} \frac{1}{15\text{km/h}} \sqrt{(x_{R_i} - x_{R_{i+1}})^2 + (y_{R_i} - y_{R_{i+1}})^2}$$

Da Lisa $(\,0\,|\,H\,)$ zum selben Zeitpunkt wie der Bus erreichen muss, gilt

$$7{:}30 + \Delta T_{\text{Bus}} = T_{\text{Lisa}} + \Delta T_{\text{Lisa}}$$

und somit

$$T_{\text{Lisa}} = 7{:}30 + \frac{H}{30\text{km/h}} - \sum_{i=1}^{L-1} \frac{1}{15\text{km/h}} \sqrt{(x_{R_i} - x_{R_{i+1}})^2 + (y_{R_i} - y_{R_{i+1}})^2}$$

Wir stehen vor folgendem Optimierungsproblem: Gesucht ist die Route R_{opt} mit maximalen T_{Lisa}, die sog. *optimale Route*. Dazu habe ich mir zuerst überlegt, wie sie verläuft, wenn es keine Hindernisse gibt.

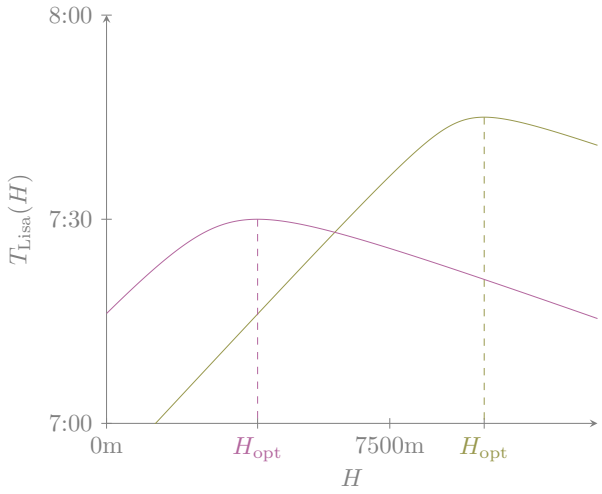
1.1.1 Optimale Route ohne Hindernisse

Wenn es keine Hindernisse gibt, kann sich Lisa von P_{start} direkt zu $(\,0\,|\,H\,)$ bewegen, da $\overline{P_{\text{start}}(\,0\,|\,H\,)}$ sicher kein Hindernis schneidet. Dies wird sie auch tun, da der kürzeste Weg von P_{start} zu $(\,0\,|\,H\,)$ genau $\overline{P_{\text{start}}(\,0\,|\,H\,)}$ ist. Somit hat die optimale Route genau 2 Eckpunkte und $L = 2$.

Deshalb ist T_{Lisa} vereinfachbar. Da $P_{start} = (\,x_{start}\,|\,y_{start}\,)$ nicht von Lisa einstellbar ist, ist H die einzige einstellbare Variable. Deshalb habe ich T_{Lisa} zusätzlich in Abhängigkeit von H gesetzt:

$$T_{\text{Lisa}}(H) = 7{:}30 + \frac{H}{30\text{ km/h}} - \frac{1}{15\text{ km/h}} \sqrt{(x_{start} - 0)^2 + (y_{start} - H)^2}$$

Nun müssen wir ein $H_{opt} \geq 0$ so wählen, dass $T_{\text{Lisa}}(H_{opt})$ maximal ist. Dafür muss H_{opt} die Stelle des globalen Maximums von T_{Lisa} im Intervall $[0;\infty)$ sein.
Die folgende Abbildung zeigt zwei beispielhafte Funktionsgraphen von T_{Lisa}:



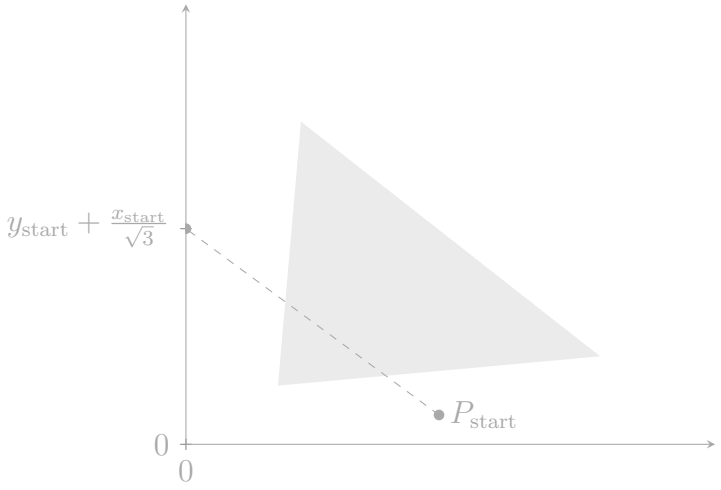
In beiden Funktionsgraphen kann man H_{opt} als die Stelle des globalen Maximums erkennen. Ich werde in Abschnitt 1.5.2 nachweisen, dass

$$H_{\text{opt}} = y_{\text{start}} + \frac{x_{\text{start}}}{\sqrt{3}}$$

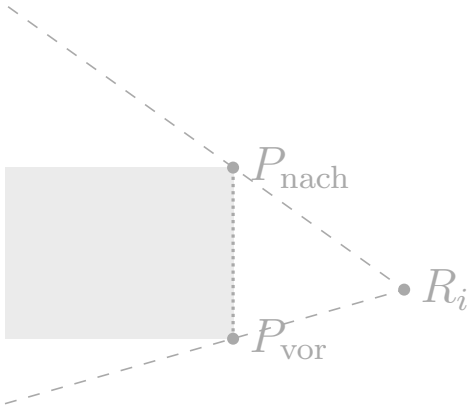
Somit verläuft die optimale Route, wenn es keine Hindernisse gibt, von P_{start} zu $(\,0\,|\,y_{\text{start}} + \frac{x_{\text{start}}}{\sqrt{3}}\,)$. Deshalb gilt generell, auch wenn es Hindernisse gibt, dass die optimale Route von $(\,x\,|\,y\,)$ zu $(\,0\,|\,y + \frac{x}{\sqrt{3}}\,)$ verläuft, wenn $\overline{(\,x\,|\,y\,)(\,0\,|\,y + \frac{x}{\sqrt{3}}\,)}$ kein Hindernis schneidet.

1.1.2 Optimale Route mit Hindernissen

Wenn es Hindernisse gibt, kann $\overline{P_{\text{start}}(\,0\,|\,y_{\text{start}} + \frac{x_{\text{start}}}{\sqrt{3}}\,)}$ ein Hindernis schneiden, in welchem Fall die optimale Route anders verlaufen muss. Ein Beispiel dafür ist in der folgenden Abbildung gezeigt:

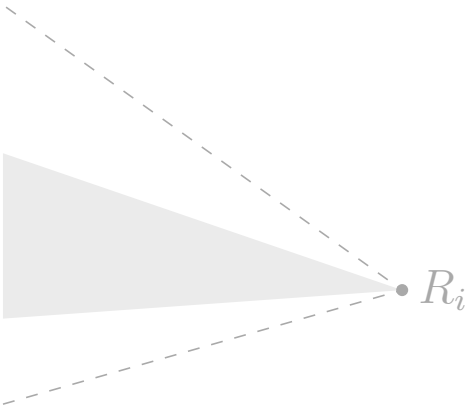


Wenn es auf einer Route einen Eckpunkt R_i, einen Punkt P_{vor}, den Lisa zeitlich vor R_i erreicht, und einen Punkt P_{nach}, den Lisa zeitlich nach R_i erreicht, gibt, sodass $\overline{P_{\text{vor}}P_{\text{nach}}}$ kein Hindernis schneidet, ist die Route nicht optimal. Dann kann Lisa sie nämlich abkürzen, indem sie sich von P_{vor} direkt zu P_{nach} bewegt, ohne sich vorher zu R_i zu bewegen. Ein Beispiel dafür ist in der folgenden Abbildung gezeigt. Die gestrichelte Linie zeigt einen Teil einer Route, die punktierte Linie zeigt, wie Lisa diesen abkürzen kann:



Deshalb schneidet in der optimalen Route $\overline{P_{\text{vor}}P_{\text{nach}}}$ für alle möglichen R_i, P_{vor}, P_{nach} ein Hindernis. Dazu muss jeweils ein Teil des Hindernisses zwischen P_{vor} und P_{nach} liegen,

ohne dass P_{vor} oder P_{nach} innerhalb des Hindernisses liegen. Da P_{vor} und P_{nach} unendlich nah an R_i und dadurch unendlich nah aneinander liegen können, muss an R_i ein unendlich dünner Teil eines Hindernisses liegen. Dazu kommen nur Ecken der Hindernispolygone in Frage.



Deshalb sind alle Eckpunkte der optimalen Route Ecken von Hindernispolygonen. Ausgenommen davon sind R_1, da es für R_1 kein P_{vor} gibt, und R_L, da es für R_L kein P_{nach} gibt.

1.2 Formulierung als kürzester-Pfad-Problem

Aus unseren Überlegungen wissen wir, dass die optimale Route von $(\,x\,|\,y\,)$

- entweder zu $(\,0\,|\,y+\frac{x}{\sqrt{3}}\,)$, wenn $\overline{(\,x\,|\,y\,)(\,0\,|\,y+\frac{x}{\sqrt{3}}\,)}$ kein Hindernis schneidet
- oder ansonsten zu einer Ecke eines Hindernispolygons, zu der die Strecke kein Hindernis schneidet

verläuft. Da Lisa für letzteres immer mehrere Ecken zur Auswahl hat, kommen mehrere Routen für die optimale Route in Frage. Diese Routen lassen sich als gerichteter kantengewichteter Graph $G=(V,E)$ modellieren, für den folgendes gilt:

- Die Menge der Knoten V enthält einen Knoten s für P_{start}, einen Knoten t für $(\,0\,|\,H\,)$ und einen Knoten für jede Ecke eines Hindernispolygons.

  Jedem Knoten u außer t ist eine Position $P_u=(\,x_u\,|\,y_u\,)$ zugewiesen, die der Position der ihm zugehörigen Ecke bzw. P_{start} entspricht.

- Die Menge der Kanten E enthält für jeden Knoten u außer t:
  - entweder wenn $\overline{P_u(\,0\,|\,y_u+\frac{x_u}{\sqrt{3}}\,)}$ kein Hindernis schneidet, eine gerichtete Kante zu t. Das Gewicht einer solchen Kante entspricht:

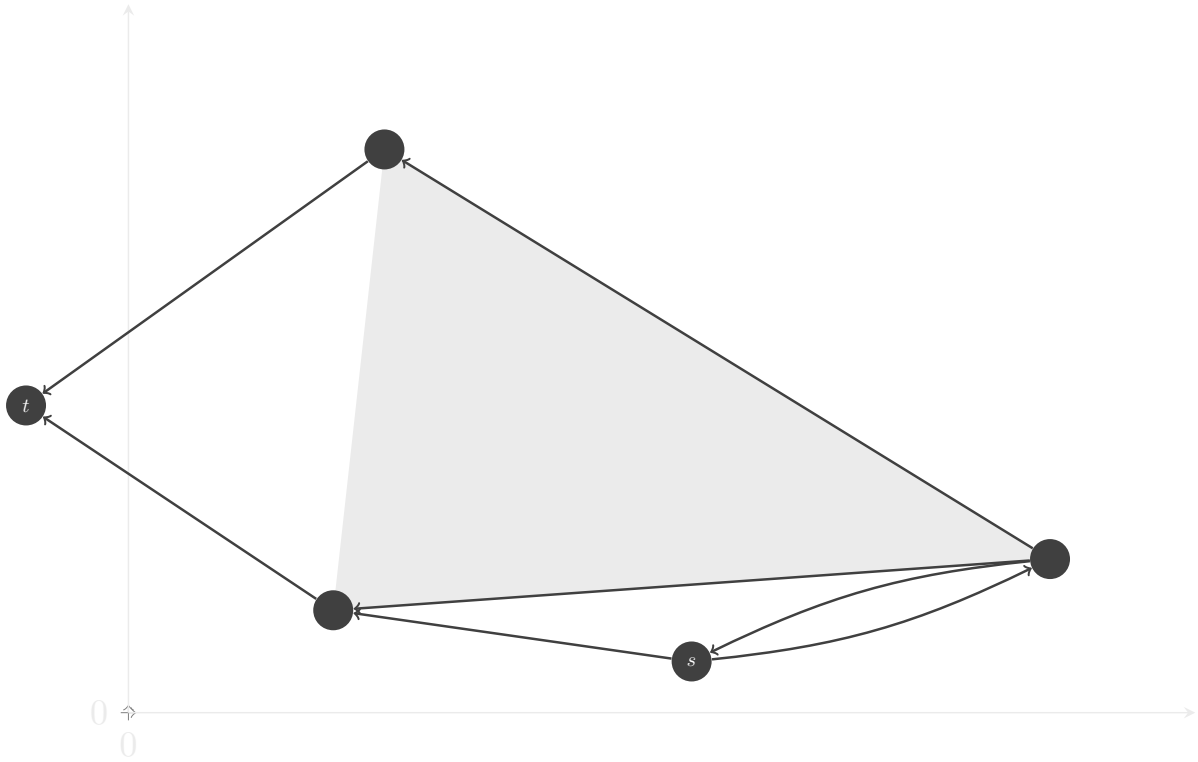
$$w(u,t)=\frac{1}{15\text{ km/h}}\sqrt{(x_u-0)^2+(y_u-(y_u+\frac{x_u}{\sqrt{3}}))^2}-(7{:}30+\frac{y_u+\frac{x_u}{\sqrt{3}}}{30\text{ km/h}})$$

  - oder ansonsten gerichtete Kanten zu jedem anderen Knoten v außer t, für den $\overline{P_uP_v}$ kein Hindernis schneidet. Das Gewicht einer solchen Kante entspricht:

$$w(u,v)=\frac{1}{15\text{ km/h}}\sqrt{(x_u-x_v)^2+(y_u-y_v)^2}$$

Jeder Pfad von s zu t entspricht einer Route. Die Kantengewichte sind so gewählt, dass die Länge des Pfades minus dem Startzeitpunkt Lisas für die Route, also $-T_{\text{Lisa}}$, entspricht. Deshalb entspricht der kürzeste Pfad von s zu t der optimalen Route.

Hier ein Beispielgraph (in dunkelgrau, zur Übersichtlichkeit ohne Kantengewichte) und die zugehörigen Hindernisse (in hellgrau):



Somit habe ich das Problem als kürzester-Pfad-Problem formuliert. Dies hat den Vorteil, dass das kürzester-Pfad-Problem ein gut untersuchtes graphentheoretisches Standardproblem ist und effiziente Algorithmen zum Finden des kürzesten Pfads existieren, die ich somit verwenden kann.

Für eine gegebene Hinderniskonstellation wird zuerst G erstellt. Wie viel Zeit wird dafür benötigt?

- mit n bezeichne ich die Anzahl der Ecken aller Hindernispolygone.
- Das Einfügen der $n + 2$ Knoten in V benötigt Zeit $\Theta(n)$.
- Das Einfügen der Kanten in E benötigt Zeit $\Omega(n^2)$, da zu prüfen, ob eine Strecke kein Hindernis schneidet, Zeit $\Theta(n)$ benötigt[1] und im besten Fall, wenn $\overline{P_u(\,0\,|\,y_u + \frac{x_u}{\sqrt{3}}\,)}$ für jeden Knoten u außer t kein Hindernis schneidet, nur $n + 1$ Prüfungen durchgeführt werden.
- Das Einfügen der Kanten in E benötigt Zeit $\mathcal{O}(n^3)$, da im schlechtesten Fall, wenn $\overline{P_u(\,0\,|\,y_u + \frac{x_u}{\sqrt{3}}\,)}$ für jeden Knoten u außer t ein Hindernis schneidet, $(n + 1)^2$ Prüfungen durchgeführt werden.

[1] Das dazu entwickelte Verfahren erkläre ich in Abschnitt 1.5.1.

Sowohl im besten als auch im schlechtesten Fall dominiert die Zeit zum Einfügen der Kanten die zum Einfügen der Knoten, sodass das Erstellen von G insgesamt Zeit $\Omega(n^2)$ und $\mathcal{O}(n^3)$ benötigt.

Da $n+2$ Knoten, im besten Fall $n+1$ Kanten und im schlechtesten Fall $(n+1)^2$ Kanten gespeichert werden, wird für G Speicher $\Omega(n)$ und $\mathcal{O}(n^2)$ benötigt.

1.3 Finden des kürzesten Pfads

Nachdem G erstellt wurde, wird der kürzeste Pfad von s zu t gefunden.

Wenn Zyklen mit negativem Gewicht, negative Zyklen, existieren, gibt es keinen kürzesten Pfad von s zu t. Dann gibt es nämlich zu jedem Pfad von s zu t einen kürzeren, der einen negativen Zyklus oft genug durchläuft.

Da nur Kanten zu t negatives Gewicht haben können und von t keine Kanten ausgehen, existieren keine Zyklen, die Kanten mit negativem Gewicht enthalten. Da negative Zyklen Kanten mit negativem Gewicht enthalten müssen, existieren keine negativen Zyklen.

Somit gibt es immer einen kürzesten Pfad von s zu t. Zum Finden dessen stehen zwei Algorithmen zur Auswahl:

1.3.1 Bellman-Ford-Algorithmus

Er arbeitet wie folgt: Zu jedem Knoten u wird von s ein Pfad geschätzt. Seine Länge, $u.d$, und der Vorgängerknoten von u in ihm, $u.\pi$, wird gespeichert. Zu Beginn des Algorithmus ist von s zu keinem anderen Knoten ein Pfad bekannt, außer der zu sich selbst mit Länge 0. Deshalb wird d für s mit 0 und für die anderen Knoten mit ∞ initialisiert, π wird für alle Knoten mit NULL initialisiert:

```
1  for u ∈ V
2      u.d = ∞
3      u.π = NULL
4  s.d = 0
```

$|V| - 1$ Durchläufe werden ausgeführt, in jedem welcher jede Kante relaxiert wird. Bei der Relaxation einer Kante (u, v) wird geprüft, ob der Pfad über den geschätzten Pfad zu u und dann über (u, v) zu v kürzer ist als der geschätzte Pfad zu v. Mathematisch formuliert, ob

$$v.d > u.d + w(u, v)$$

Wenn dies der Fall ist, wurde eine Abkürzung gefunden und der geschätzte Pfad zu v wird verkürzt, wofür $v.d$ zu $u.d + w(u, v)$ und $v.\pi$ zu u aktualisiert wird.

```
5  repeat |V| − 1 times
6      for (u, v) ∈ E
7          if v.d > u.d + w(u, v)
8              v.d = u.d + w(u, v)
9              v.π = u
```

Die geschätzten Pfade werden somit immer weiter verkürzt, bis sie nach dem $(|V|-1)$-ten Durchlauf den kürzesten Pfaden entsprechen. Danach wird der kürzeste Pfad von s zu t konstruiert, indem von t den Vorgängerknoten gefolgt wird, bis s erreicht wird.

Wie viel Zeit benötigt der Algorithmus?

- Zeilen 1-4 benötigen Zeit $\Theta(|V|)$.
- Zeilen 5-9 benötigen Zeit $\Theta(|V| \cdot |E|)$
- Die Rekonstruktion des kürzesten Pfads benötigt Zeit $\mathcal{O}(|V|)$, da dieser aus maximal $|V|-1$ Kanten besteht. Bestünde er aus mehr Kanten, würden Knoten mehrmals auf ihm liegen. Dies ist nicht möglich, da man dann Teilpfade zwischen zwei Vorkommen des selben Knotens entfernen könnte und einen kürzeren Pfad erhalten würde.

Insgesamt benötigt der Algorithmus somit Zeit $\Theta(|V| \cdot |E|)$. Da $|V| = n + 2$, im besten Fall $|E| = n + 1$ und im schlechtesten Fall $|E| = (n+1)^2$, entspricht dies $\Omega(n^2)$ und $\mathcal{O}(n^3)$.

Da lediglich für jeden der $n+2$ Knoten d- und π-Wert gespeichert werden, benötigt der Algorithmus Speicher $\Theta(n)$.

1.3.2 Dijkstras Algorithmus

Er ist ist sehr bekannt, oft die beste Wahl, um kürzeste Pfade in kantengewichteten Graphen zu finden und arbeitet wie folgt: Die Initialisierung gleicht der des Bellman-Ford-Algorithmus, außer dass auch eine Menge S verwaltet wird, die die Knoten enthält, zu denen der kürzeste Pfad noch nicht gefunden wurde. Bei der Initialisierung sind dies alle außer s.

1 **for** $u \in V$
2 $\quad u.d = \infty$
3 $\quad u.\pi = \text{NULL}$
4 $s.d = 0$
5 $S = V \setminus \{s\}$

Mehrere Durchläufe werden ausgeführt, jeder welcher wie folgt abläuft: Der kürzeste Pfad zu einem weiteren Knoten $x \in S$ wird gefunden, was heißt, dass der geschätzte Pfad zu x sicher der kürzeste Pfad ist und x aus S entfernt wird. Die Grundidee ist, dass Pfade zunehmender Länge gefunden werden, also x der Knoten ist, zu dem der kürzeste Pfad kürzer als zu jedem anderen Knoten in S ist. Dies erleichtert das Finden der kürzesten Pfade:

Daraus folgt nämlich $x.\pi \in V \setminus S$, da wenn $x.\pi \in S$, der Pfad zu $x.\pi$ länger als der zu x sein muss, wofür $w(x.\pi, x) < 0$ sein muss, der Algorithmus jedoch annimmt, dass alle Kantengewichte nichtnegativ sind.

Damit der geschätzte Pfad zu x sicher der kürzeste Pfad ist, müssen deshalb vor dem Durchlauf alle (u, v), wobei $u \in V \setminus S$, $v \in S$, jeweils nachdem der kürzeste Pfad zu u gefunden wurde, relaxiert worden sein. Diese Invariante wurde bei der Initialisierung

geltend gemacht, indem alle von s ausgehenden Kanten relaxiert wurden[2], und bleibt geltend, wozu am Ende des Durchlaufs alle von x ausgehenden Kanten relaxiert werden.

Nachdem der kürzeste Pfad zu x gefunden wurde, wird seine Länge/d-Wert nicht niedriger. Entscheidend ist, dass in S kein Knoten einen niedrigeren d-Wert als x haben darf, da sonst der kürzeste Pfad zu x nicht kürzer als zu jedem anderen Knoten in S ist. Da es x immer gibt, ist x somit der Knoten mit minimalem d-Wert.

Damit x schnell gefunden werden kann, wird S durch eine Min-Prioritätswarteschlange dargestellt, wobei die d-Werte als Schlüssel dienen.

Es werden so viele Durchläufe ausgeführt, bis zu jedem Knoten der kürzeste Pfad gefunden wurde, also S leer ist.

```
 6  while S ≠ ∅
 7                                  // Entfernen und Zurückgabe
 8      x = EXTRACT-MIN(S)          // des Knotens mit minimalem d-Wert
 9      for (x, v) ∈ E
10          if x.d + w(x, v) < v.d
11              v.d = x.d + w(x, v)
12              v.π = x
```

Wenn die Prioritätswarteschlange mit einem Fibonacci- oder Relaxierten Heap implementiert wird, wird eine Laufzeitkomplexität von $\mathcal{O}(|E| + |V| \log |V|)$ erreicht, was $\mathcal{O}(n^2)$ entspricht.

1.3.3 Wahl des Algorithmus

Nach meinem Leitprinzip „Simple is smart“ habe ich den Bellman-Ford-Algorithmus gewählt, da er um einiges einfacher ist. Das liegt auch an folgenden Umständen:

- Er ist korrekt, auch wenn negative Kantengewichte existieren. Dijkstras Algorithmus ist dies nicht, da er nicht berücksichtigt, dass dann $x.\pi \in S$ möglich ist. Da die Kanten zu t negatives Gewicht haben können, ist es komplizierter, diesen auf G anzuwenden: Erst müsste er auf G ohne die Kanten zu t angewendet werden, danach müssten seperat die Kanten zu t relaxiert werden.
- Er iteriert nur in beliebiger Reihenfolge über die Kanten, wofür eine Kantenliste(*edge list*) ausreicht. Dijkstras Algorithmus greift auf von einzelnen Knoten ausgehende Kanten zu, wofür G mit Adjazenzlisten repräsentiert werden sollte, was komplizierter ist.

Wie wir im nächsten Abschnitt sehen werden, ist der Bellman-Ford-Algorithmus ausreichen effizient. Dass Dijkstras Algorithmus effizienter ist, bringt für die Aufgabe keinen Vorteil, da es die Laufzeit des Lösungsverfahrens höchstens um einen kleinen konstanten Faktor verringert und die Laufzeitkomplexität gleich bleibt.

[2]dies wurde im Pseudocode weggelassen

1.4 Qualität des Verfahrens

Insgesamt ist mein Lösungsverfahren:

1. Erstellen des Graphen
2. Finden des kürzesten Pfads mit dem Bellman-Ford-Algorithmus

Die Laufzeit setzt sich aus den Laufzeiten der beiden Teilschritte zusammen, weshalb das Lösungsverfahren eine Laufzeitkomplexität von $\Omega(n^2)$ und $\mathcal{O}(n^3)$ hat. Da sie niedriggradig polynomiell ist, halte ich es für ausreichen effizient. Die hohe Effizienz wird empirisch dadurch belegt, dass es jedes der vorgegebenen und der von mir sorgsam von Hand erstellten Beispiele in wenigen Millisekunden gelöst hat. Selbst sehr große Beispiele hat es in annehmbarer Zeit gelöst.

Aus dem für den Graphen und für den Bellman-Ford-Algorithmus benötigten Speicher setzt sich die Speicherkomplexität von $\Omega(n)$ und $\mathcal{O}(n^2)$ des Lösungsverfahrens zusammen. Sie ist für den besten Fall optimal. Auch für den schlechtesten Fall halte ich sie für ausreichend gering.

1.5 Anhang

1.5.1 Prüfen nach Schneiden von Hindernissen

Um zu prüfen, ob eine Strecke kein Hindernis schneidet, will ich jede der Seiten der Polygone einem folgender Fälle zuordnen:

Fall 1 Strecke parallel zu Seite → kein Schnitt[3]

Fall 2 Seite nicht parallel zu Strecke

- **Fall 2.1** Strecke berührt Seite nicht → kein Schnitt
- **Fall 2.2** Strecke berührt Seite zwischen Endpunkten → Schnitt[4]
- **Fall 2.3** Strecke berührt Seite an Endpunkt
  - **Fall 2.3.1** Strecke innerhalb Polygon fortgeführt → Schnitt
  - **Fall 2.3.2** Strecke nicht innerhalb Polygon fortgeführt → kein Schnitt

Wenn bei allen Seiten kein Schnitt vorliegt, schneidet die Strecke kein Hindernis, ansonsten schon. Zum Zuordnen wird zuerst für die Strecke(mit Endpunkten P_1, P_2) und die Seite(mit Endpunkten Q_1, Q_2) die Geradengleichung bestimmt:

$$y = m_P x + c_P, \text{ wobei } m_P = \frac{y_{P_2} - y_{P_1}}{x_{P_2} - x_{P_1}},\ c_P = y_{P_1} - m_P x_{P_1}$$
$$y = m_Q x + c_Q, \text{ wobei } m_Q = \frac{y_{Q_2} - y_{Q_1}}{x_{Q_2} - x_{Q_1}},\ c_Q = y_{Q_1} - m_P x_{Q_1}$$

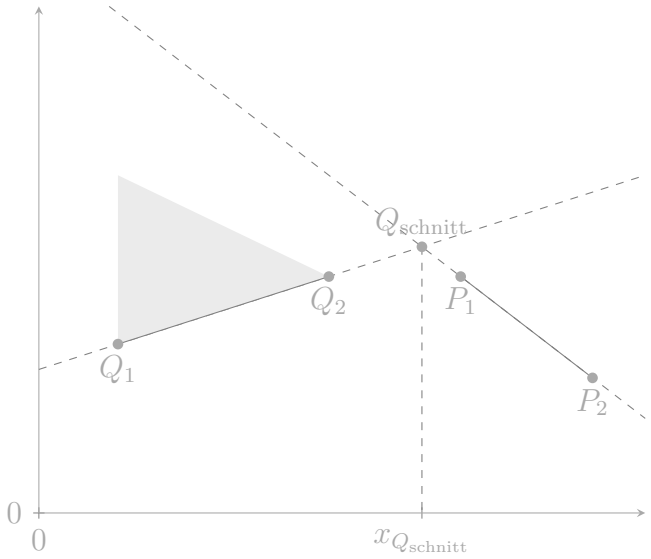
[3] Da wenn sie aufeinanderliegen die Strecke als unendlich nah und doch neben der Seite außerhalb des Polygons liegend gesehen werden kann.

[4] Damit kein Schnitt vorläge, müsste dort die Strecke enden, also eine Polygonecke liegen. Das ist nicht möglich, wenn Hindernispolygone sich nicht berühren und nicht überlappen.

Wenn $m_P = m_Q$, gilt Fall 1. Sonst wird der x-Wert des Schnittpunkts Q_{schnitt} der Geraden bestimmt:

$$x_{Q_{\text{schnitt}}} = \frac{c_Q - c_P}{m_P - m_Q}$$

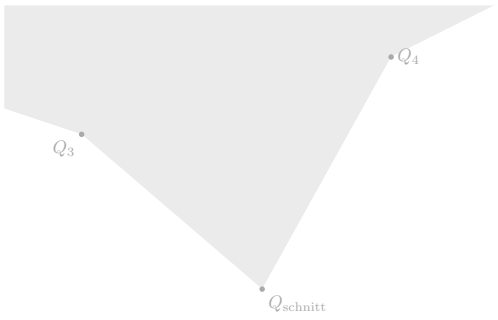
Wenn Q_{schnitt} nicht auf der Strecke oder der Seite liegt, also $x_{Q_{\text{schnitt}}} \notin [x_{P_1}; x_{P_2}]$ oder $x_{Q_{\text{schnitt}}} \notin [x_{Q_1}; x_{Q_2}]$, gilt Fall 2.1.



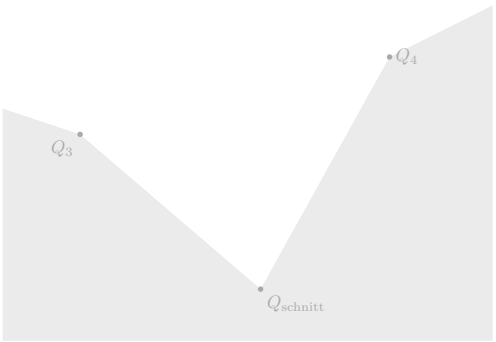
Wenn Q_{schnitt} auf der Seite liegt und kein Endpunkt ist, also $x_{Q_{\text{schnitt}}} \in (x_{Q_1}; x_{Q_2})$, gilt Fall 2.2.

Sonst ist Q_{schnitt} ein Endpunkt der Seite. Die anderen Endpunkte der beiden anliegenden Seiten werden mit Q_3 und Q_4 bezeichnet. Folgendes muss geprüft werden, um dann zwischen den Fällen 2.3.1 und 2.3.2 zu unterscheiden:

- ob die dort liegende Polygonecke konvex oder konkav ist.



konvex



konkav

Sie ist konvex, wenn $\overline{Q_3 Q_{\text{schnitt}}}$ und $\overline{Q_{\text{schnitt}} Q_4}$ an Q_{schnitt} in Richtung des Inneren des Polygons abbiegen. Das ist der Fall, wenn der Term[5]

$$(x_{Q_4} - x_{Q_3})(y_{Q_{\text{schnitt}}} - y_{Q_3}) - (x_{Q_{\text{schnitt}}} - x_{Q_3})(y_{Q_4} - y_{Q_3})$$

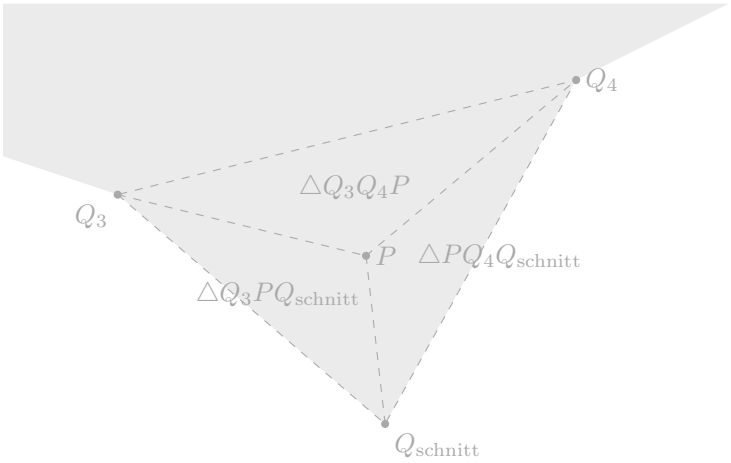
[5]Dieser ist abgewandelt von [CLRS], Kapitel 33.1.

negativ ist und das Innere sich bezüglich $\overrightarrow{Q_3 Q_{\text{schnitt}}}$ entgegen dem Uhrzeigersinn befindet, oder wenn er positiv ist und es sich im Uhrzeigersinn befindet.[6]

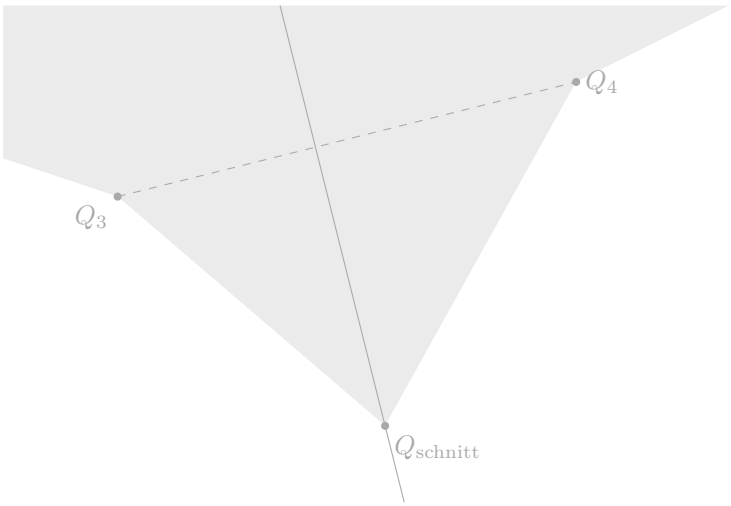
- ob die Strecke $\overline{Q_3 Q_4}$ schneidet, was ähnlich geschieht, außer dass nicht zwischen den Fällen 2.2, 2.3.1 und 2.3.2 unterschieden wird.
- ob $P = P_1$ und $P = P_2$ im Dreieck $\triangle Q_3 Q_4 Q_{\text{schnitt}}$ liegen. Dafür wird seine Fläche A mit der Formel[MOR]

$$A = |\frac{x_{Q_3}(x_{Q_{\text{schnitt}}} - x_{Q_4}) + x_{Q_{\text{schnitt}}}(x_{Q_4} - x_{Q_3}) + x_{Q_4}(x_{Q_3} - x_{Q_{\text{schnitt}}})}{2}|$$

berechnet, und ebenso die Flächen A_1, A_2, A_3 der Teildreiecke $\triangle P Q_4 Q_{\text{schnitt}}$, $\triangle Q_3 P Q_{\text{schnitt}}$, $\triangle Q_3 Q_4 P$. Wenn $A = A_1 + A_2 + A_3$ und $A_1, A_2, A_3 \neq 0$, liegt P im Dreieck.[7]



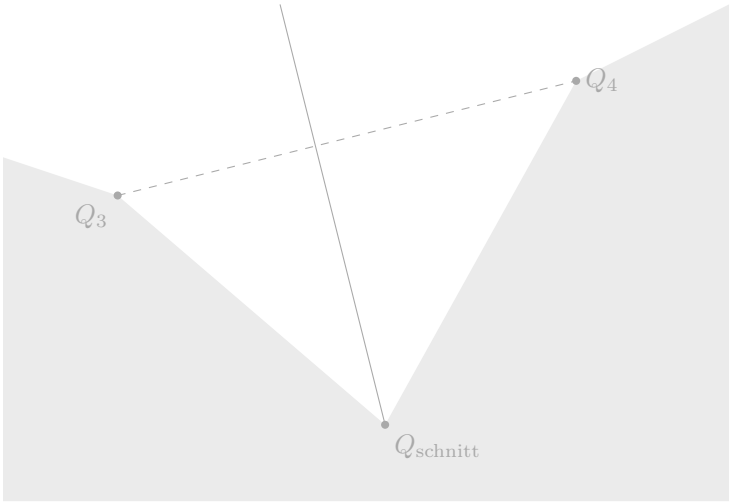
Wenn die Ecke konvex ist, gilt Fall 2.3.1, wenn die Strecke $\overline{Q_3 Q_4}$ schneidet oder P_1 oder P_2 im Dreieck liegen. Sonst gilt Fall 2.3.2.



[6] Dies wird für jede Seite vorberechnet. Bei der Vorberechnung muss man wissen, ob die Ecken des Polygons entgegen oder im den Uhrzeigersinn angegeben wurden, wofür dessen signierte Fläche[WMW] mit der gaußschen Trapezformel berechnet und geprüft wird, ob diese positiv oder negativ ist.

[7] Dieses Vorgehen ist abgewandelt von [GFG].

Wenn die Ecke konvkav ist, gilt Fall 2.3.2, wenn Q_{schnitt} ein Endpunkt der Strecke ist und sie entweder $\overline{Q_3Q_4}$ schneidet, oder ihr anderer Endpunkt im Dreieck liegt. Sonst gilt Fall 2.3.1.



Das Zuordnen einer Seite benötigt Zeit $\Theta(1)$, da maximal eine konstante Anzahl an Berechnungen durchgeführt wird, und n Seiten werden zugeordnet, sodass insgesamt Zeit $\Theta(n)$ benötigt wird.[8]

Es ist typisch für die algorithmische Geometrie, dass selbst einfache Aufgaben kompliziert zu lösen sind und es viele Spezialfälle gibt, die seperat behandelt werden müssen. Die Qualität des Vorgehens sehe ich deshalb darin, dass es mit wenigen Fällen auskommt und diese relativ einfach, elegant und mit wenigen Berechnungen unterscheiden kann.

Der einzige Spezialfall ist, wenn Strecken oder Seiten senkrecht sind, sodass bei der Berechnung von m durch 0 geteilt würde. Das Vorgehen für diesen ähnelt dem normalen, weshalb es nicht im Detail beschrieben wird.

1.5.2 Nachweis H_{opt}

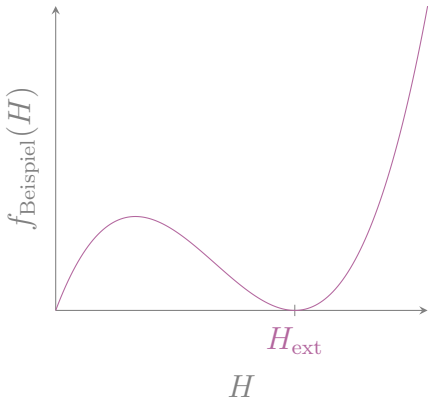
Um H_{opt} zu finden, muss man nachweisen, dass T_{Lisa} im Intervall $[0;\infty)$ ein globales Maximum hat.

Dafür muss man den Fall ausschließen, dass T_{Lisa} an H_{ext}, der Extremstelle mit dem größten H-Wert, ein lokales Minimum hat. In diesem Fall würden die Funktionswerte von T_{Lisa} nach H_{ext} nur zunehmen, sodass T_{Lisa} streng monoton zunehmend im Intervall $[H_{\text{ext}};\infty)$ wäre.

T_{Lisa} könnte nach H_{ext} nämlich nicht abnehmen, da dafür, obwohl keine Extremstelle mit einem größeren H-Wert als H_{ext} existiert, eine Extremstelle mit einem größeren H-Wert als H_{ext} existieren müsste, an der T_{Lisa} vom Zunehmen zum Abnehmen wechselt.

[8] Genau genommen $\Omega(1)$, indem sobald bei einer Seite ein Schnitt vorliegt, keine weiteren Seiten zugeordnet werden.

Da T_{Lisa} deshalb ins Unendliche ansteigt, gäbe es zu jedem Funktionswert einen größeren, weshalb T_{Lisa} im Intervall $[0;\infty)$ kein globales Maximum hätte. Hier der Graph einer Beispielfunktion, für die der Fall eintritt:



Der Fall ist ausgeschlossen, wenn T_{Lisa} an H_{ext} ein lokales Maximum hat. Dann ist H_{opt} die Stelle mit dem größten Funktionswert aus allen Extremstellen im Intervall und der unteren Intervallgrenze 0.[9]

Zur Erinnerung:

$$T_{\text{Lisa}}(H) = 7{:}30 + \frac{H}{30\ \text{km/h}} - \frac{1}{15\ \text{km/h}}\sqrt{(x_{start}-0)^2 + (y_{start}-H)^2}$$

Als erstes habe ich die erste Ableitung von T_{Lisa} gebildet, wozu ich die Kettenregel angewendet habe:

$$T'_{\text{Lisa}}(H) = \frac{1}{30\ \text{km/h}} - \frac{1}{15\ \text{km/h}} \cdot \frac{H - y_{\text{start}}}{\sqrt{(x_{start}-0)^2 + (y_{start}-H)^2}}$$

Durch Gleichsetzen mit 0, Umformen zu einer quadratischen Funktion, Anwenden der abc-Formel und Vereinfachen habe ich die einzigen beiden Extremstellen, H_1 und H_2, gefunden:

$$H_1 = y_{\text{start}} + \frac{x_{\text{start}}}{\sqrt{3}} \qquad\qquad H_2 = y_{\text{start}} - \frac{x_{\text{start}}}{\sqrt{3}}$$

Da ich $x, y \geq 0$ angenommen habe, $H_1 \geq H_2$, sodass $H_1 = H_{\text{ext}}$.
Als nächstes habe ich die zweite Ableitung gebildet:

$$T''_{\text{Lisa}}(H) = -\frac{1}{15\ \text{km/h}} \cdot \frac{x_{start}^2}{((x_{start}-0)^2 + (y_{start}-H)^2)^{3/2}}$$

Durch Umformen habe ich unter der Annahme $x, y \geq 0$ gezeigt, dass $T''_{\text{Lisa}}(H_1) < 0$, sodass T_{Lisa} nach dem hinreichenden Kriterium für Extremstellen an H_1 ein lokales Maximum hat. Somit habe ich nachgewiesen, dass T_{Lisa} ein globales Maximum hat.

H_{opt} ist somit H_1, H_2 oder 0. Ich habe unter der Annahme $x, y \geq 0$[10] jeweils durch Umformen gezeigt, dass $T_{\text{Lisa}}(H_1) > T_{\text{Lisa}}(H_2)$ und $T_{\text{Lisa}}(H_1) > T_{\text{Lisa}}(0)$. Somit ist H_1 die Stelle des globalen Maximums, sodass

$$H_{\text{opt}} = y_{\text{start}} + \frac{x_{\text{start}}}{\sqrt{3}}$$

[9] Dieses Vorgehen ist abgewandelt von [CPL].
[10] Genau genommen musste ich auch annehmen, dass $(x_{start} \,|\, y_{start}) \neq (0 \,|\, 0)$

Literatur

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms, Third Edition, ISBN 0262033844 9780262033848, Kapitel 24.1 beschäftigt sich mit dem Bellman-Ford-Algorithmus, Kapitel 24.3 mit Dijkstras Algorithmus

[WMW] Wolfram-Mathworld-Artikel zur signierten Fläche von Polygonen, `http://mathworld.wolfram.com/PolygonArea.html`

[MOR] Formel zur Berechnung der Fläche von Dreiecken, `https://www.mathopenref.com/coordtrianglearea.html`

[GFG] GeeksforGeeks-Artikel zu einem Verfahren zum Prüfen, ob ein Punkt in einem Dreieck liegt, `https://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/`

[CPL] Artikel zum Finden globaler Maxima in geschlossenen Intervallen, `http://www.math.ubc.ca/~CLP/CLP1/clP_2_dc/ssec_find_maxmin.html`

2 Umsetzung

Das Lösungsverfahren wurde prozedural in C++11 implementiert. Knoten, Kanten und Seiten werden durch `structs` repräsentiert, V, E und die Menge S der Seiten werden als dynamische Arrays dargestellt, wozu `std::vector` verwendet wird. Benötigte Methoden sind:

- `eingabe()` liest die Daten aus der Standardeingabe und fügt die Knoten in V und die Seiten in S ein. Dabei werden für jede Seite auch die Eckpunkte der beiden anliegenden Seiten gespeichert, und ob sich das Innere ihr bezüglich im Uhrzeigersinn befindet, wozu für jedes Polygon `imSinn()` aufgerufen wird.
- `imSinn()` prüft, ob die Eckpunkte eines gegebenen Polygons im Uhrzeigersinn angegeben wurden.
- `graphErstellen()` fügt die Kanten in E ein, wozu häufig `schneidetHindernis()` aufgerufen wird. Die Zeiten, denen die Kantengewichte entsprechen, werden dabei in Sekunden nach 0.00 Uhr angegeben.
- `schneidetHindernis()` prüft, ob eine Strecke ein Hindernis schneidet. Die Zuordnung ist in `schneidetSeite()` implementiert, wobei häufig aufgerufen wird:
  - `schneidetStrecke()` prüft, ob sich zwei Strecken schneiden.
  - `imDreieck()` prüft, ob ein Punkt in einem Dreieck liegt.

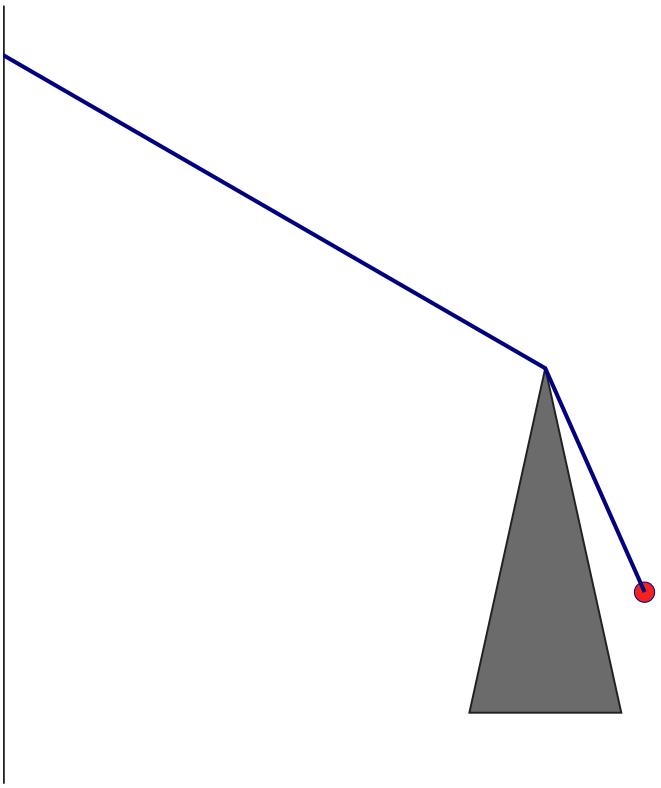
  Um Gleitkomma-Präzisionsfehler zu vermeiden, werden für alle Positionsangaben `long doubles` verwendet und diese zum Vergleichen auf 3 Nachkommastellen gerundet.
- `kuerzesterPfad()` implementiert den Bellman-Ford-Algorithmus.
- `ausgabe()` rekonstruiert die optimale Route und gibt sie aus.

3 Beispiele

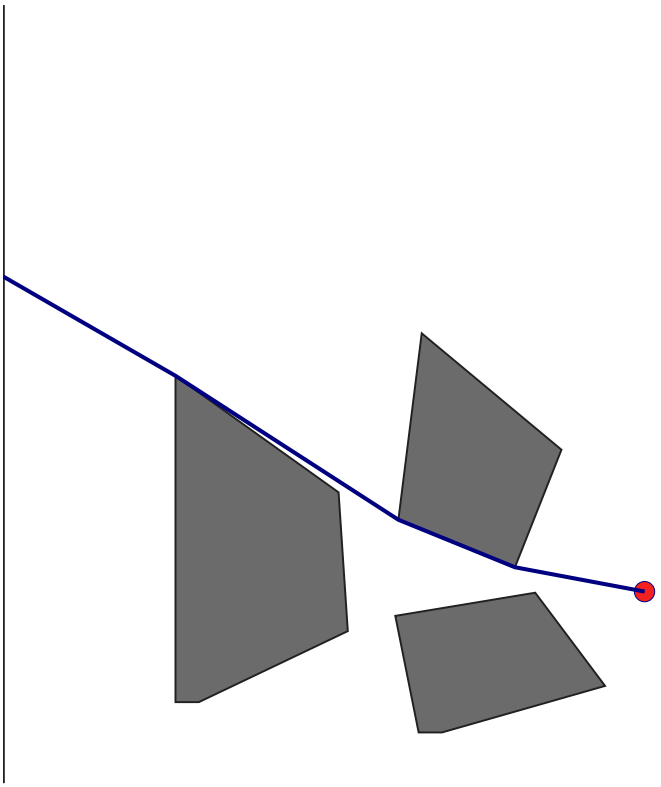
Zuerst die Beispiele von den BWINF-Webseiten:

Beispiel	Startzeit	Zielzeit	Dauer	Länge	y-Koordinate	Laufzeit des Programms
1	7:28:00	7:31:26	0:03:26	860m	719m	0ms
2	7:28:09	7:31:00	0:02:51	713m	500m	1ms
3	7:27:29	7:30:56	0:03:27	863m	464m	8ms
4	7:26:56	7:31:59	0:05:03	1263m	992m	24ms
5	7:27:55	7:30:41	0:02:46	691m	340m	24ms

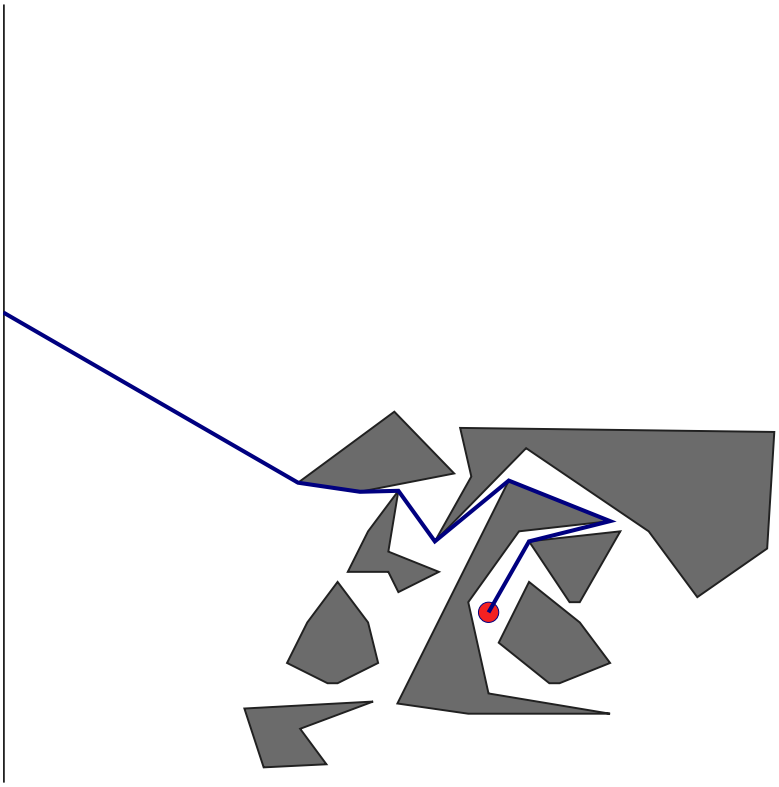
Die 3. Koordinate entspricht im Folgenden der ID.



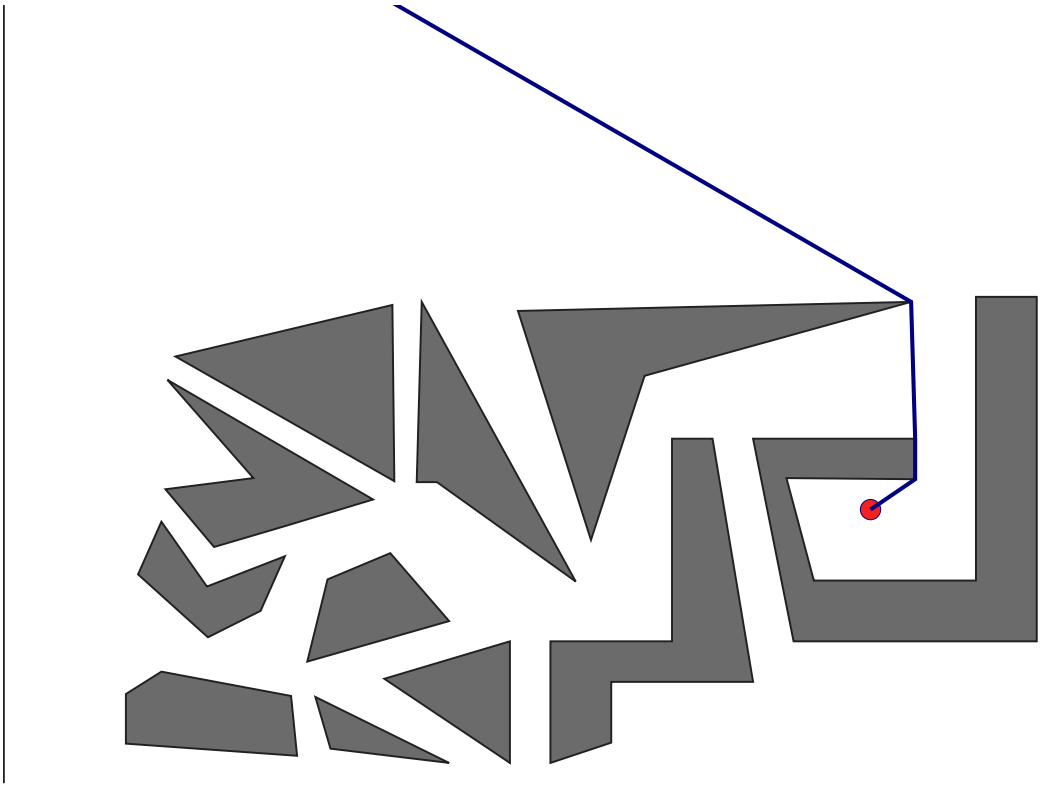
$$R_{\text{opt}} = \langle(\,633\,|\,189\,|\,\text{L}\,),(\,535\,|\,410\,|\,\text{P1}\,),(\,0\,|\,719\,|\,\text{Y}\,)\rangle.$$



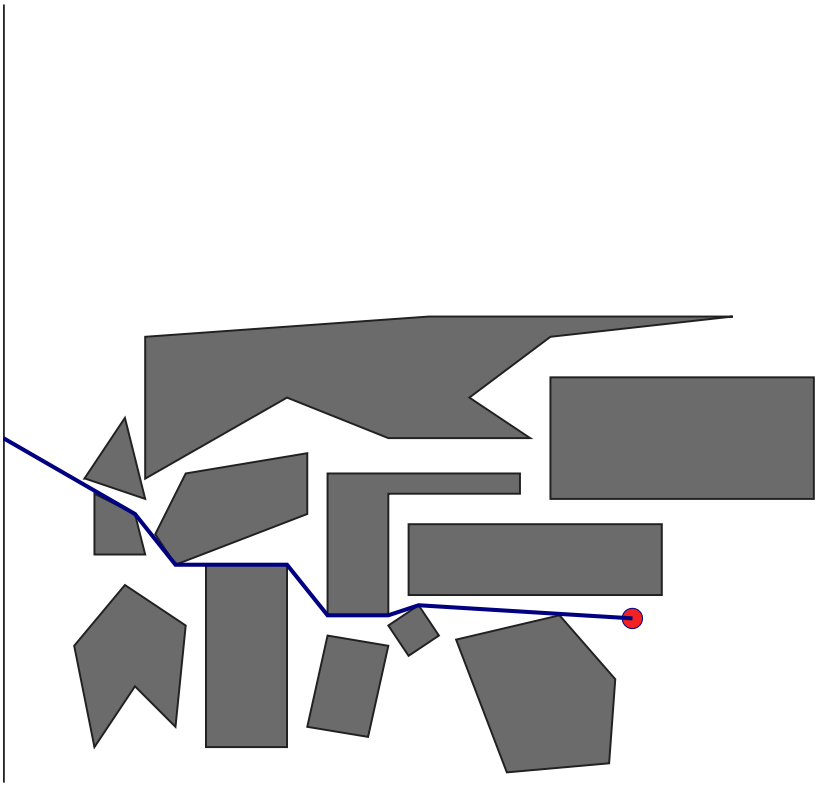
$R_{\text{opt}} = \langle (\,633\,|\,189\,|\,\text{L}\,),(\,505\,|\,213\,|\,\text{P1}\,),(\,390\,|\,260\,|\,\text{P1}\,),(\,170\,|\,402\,|\,\text{P3}\,),(\,0\,|\,500\,|\,\text{Y}\,)\rangle.$



$R_{\text{opt}} = \langle (\,479\,|\,168\,|\,\text{L}\,),(\,519\,|\,238\,|\,\text{P2}\,),(\,599\,|\,258\,|\,\text{P3}\,),(\,499\,|\,298\,|\,\text{P3}\,),(\,426\,|\,238\,|\,\text{P8}\,),$
$(\,390\,|\,288\,|\,\text{P5}\,),(\,352\,|\,287\,|\,\text{P6}\,),(\,291\,|\,296\,|\,\text{P6}\,),(\,0\,|\,464\,|\,\text{Y}\,)\rangle.$

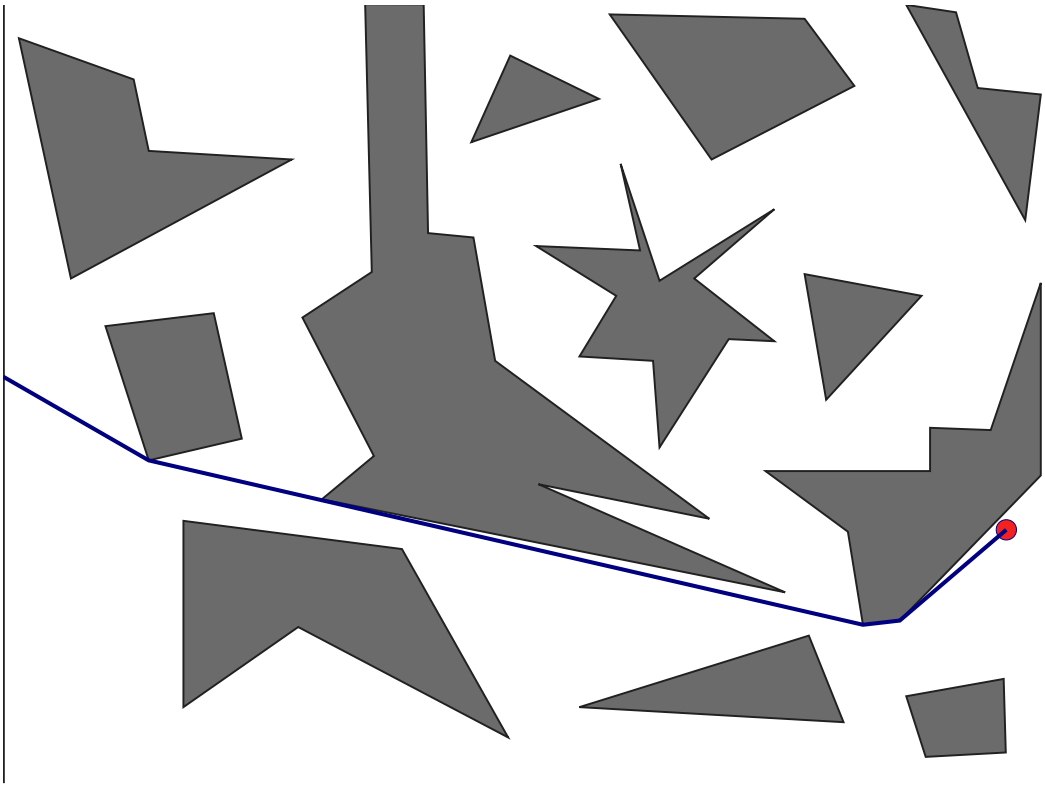


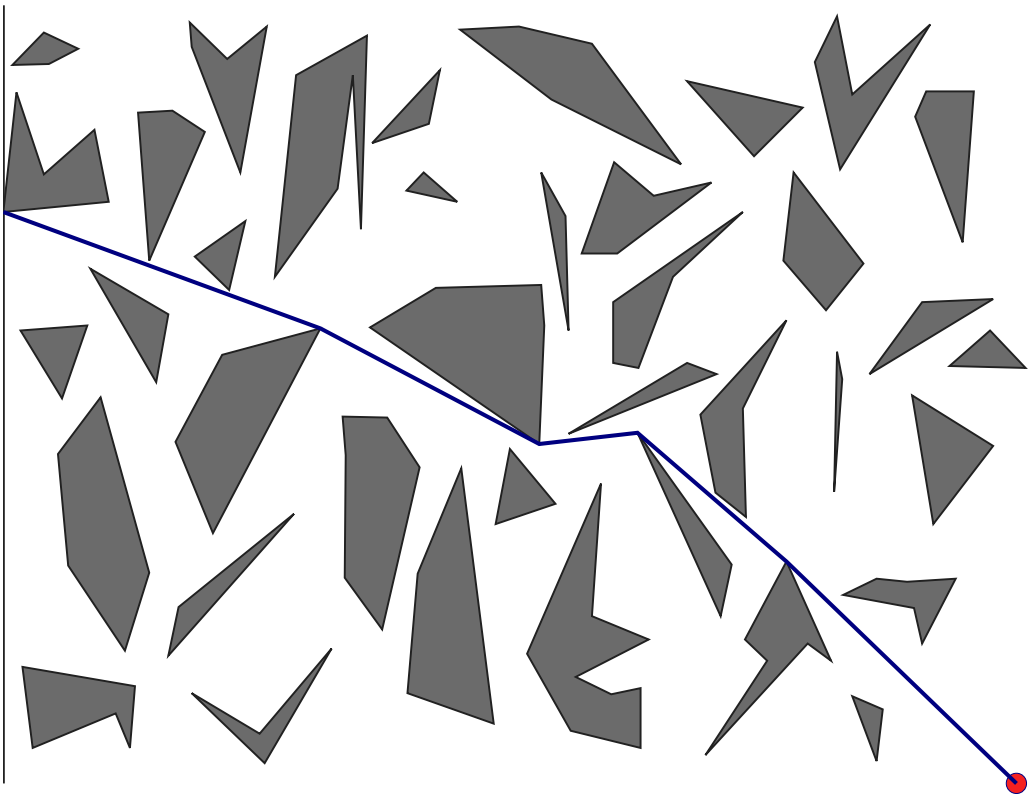
$R_{\text{opt}} = \langle(\,856\,|\,270\,|\,\text{L}\,),(\,900\,|\,300\,|\,\text{P11}\,),(\,900\,|\,340\,|\,\text{P11}\,),(\,896\,|\,475\,|\,\text{P10}\,),(\,0\,|\,992\,|\,\text{Y}\,)\rangle$.

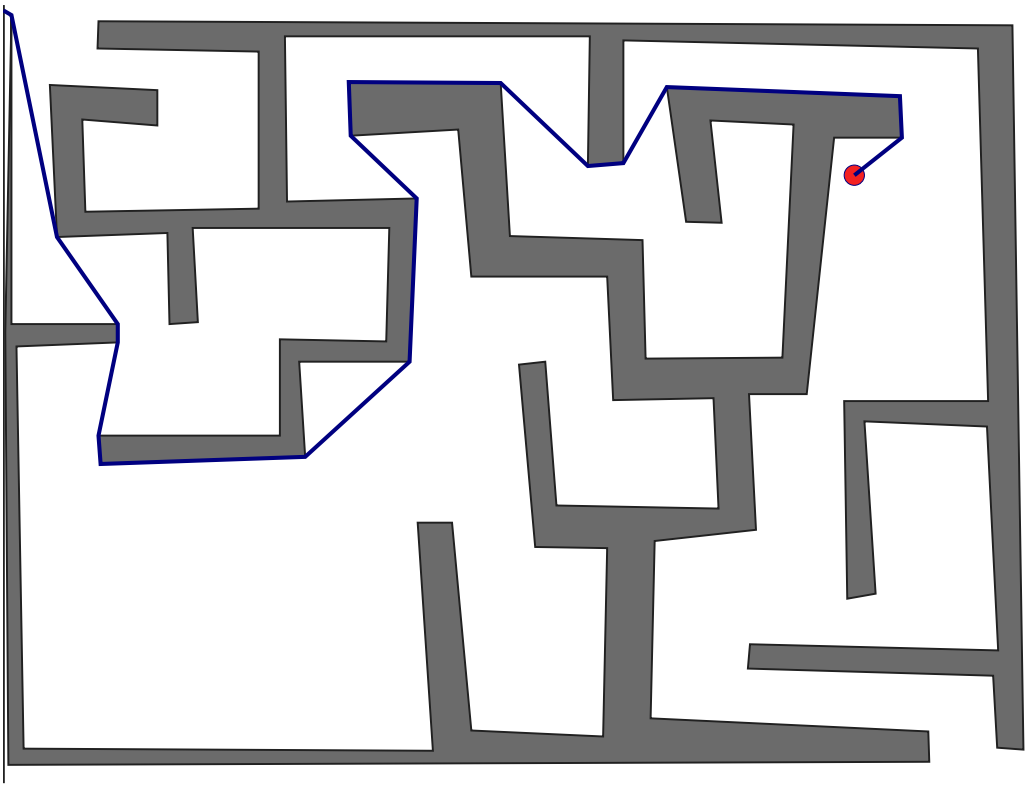


$R_{\text{opt}} = \langle(\,621\,|\,162\,|\,\text{L}\,),(\,410\,|\,175\,|\,\text{P8}\,),(\,380\,|\,165\,|\,\text{P3}\,),(\,320\,|\,165\,|\,\text{P3}\,),(\,280\,|\,215\,|\,\text{P5}\,),$
$(\,170\,|\,215\,|\,\text{P6}\,),(\,130\,|\,265\,|\,\text{P9}\,),(\,0\,|\,340\,|\,\text{Y}\,)\rangle$.

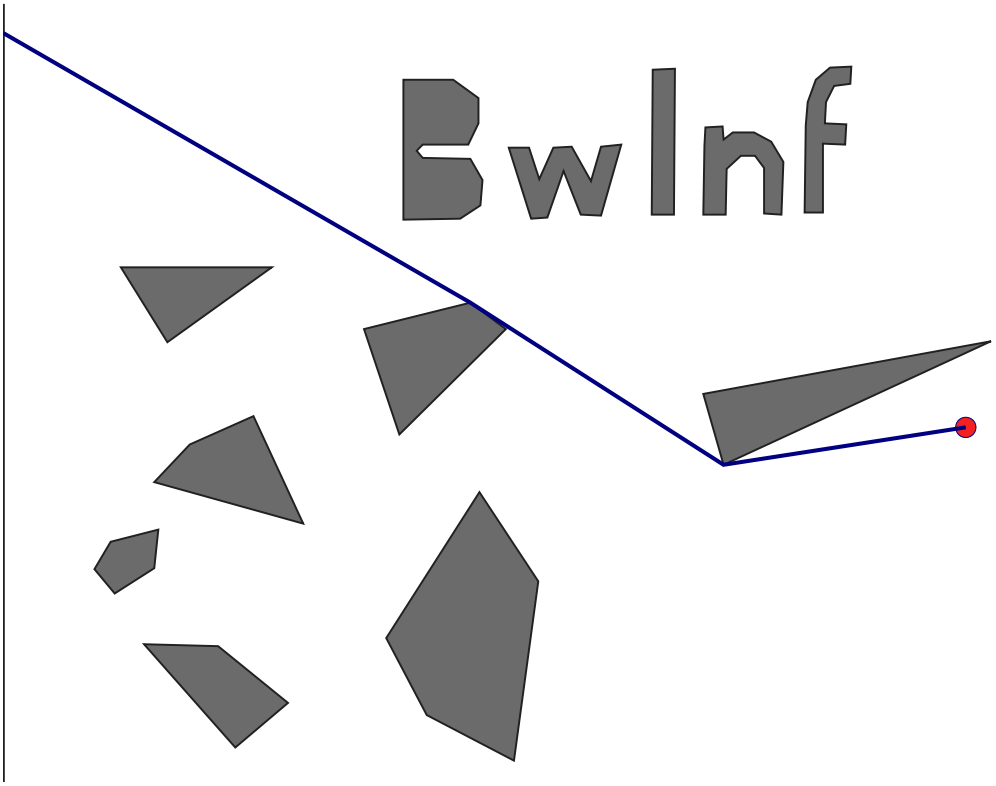
Nun folgen eigene Beispiele:







Und als letztes:



4 Quellcode

Hauptmethode

```
int main() {
    eingabe();
    graphErstellen();
    kuerzesterPfad();
    ausgabe();
    return 0;
}
```

```
typedef long double ld;

bool imUhrzeigersinn(vector<knoten> pgon, int ecken) {
    ld signierte = 0; // Flaeche
    for(int I = 0; I < ecken; I++)
        // gausssche Trapezformel
        signierte += (pgon[(I+1)\%ecken].x - pgon[I].x) *
                     (pgon[(I+1)\%ecken].y + pgon[I].y);
    return signierte > 0;
}
```

Erstellen des Graphen

```
void graphErstellen() {
    // Knoten schon eingefuegt
    // Kanten       einfuegen
    for(int u = 0; u <= N; u++) {
        if(!schneidetHindernis(V[u].x,  V[u].y, 0,
                               V[u].y + V[u].x/sqrt(3)))
            E.push_back({ u, t, w(u,t) }); // Kante zu t
        else {
            for(int v = 0; v <= N; v++)
                if(u != v && !schneidetHindernis(V[u].x, V[u].y,
                                                 V[v].x, V[v].y))
                    E.push_back({ u, v, w(u,v) }); // Kanten zu
                                                   // anderen Knoten
        }
    }
}

ld w(int u, int v) {
    if(v == t)
        return abstand(V[u].x,  V[u].y, 0,
                       V[u].y + V[u].x/sqrt(3)) / (15/3.6) - 27000 -
                      (V[u].y + V[u].x/sqrt(3)) / (30/3.6);
    else
        return abstand(V[u].x, V[u].y, V[v].x, V[v].y) / (15/3.6);
}
// euklidischer
ld abstand(ld x1, ld y1, ld x2, ld y2) {
    return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
}
```

Finden des kürzesten Pfads mit dem Bellman-Ford-Algorithmus

```
void kuerzesterPfad() {
    V[t].d = INF, V[t].pi = -1;    // fuer alle ausser t
                                   // schon initialisiert
    for(int I = 0; I <= N; I++) { // |V|-1 Durchlaeufe
        for(kante e : E) {
            // relaxieren
            if (V[e.v].d  > V[e.u].d+e.w) {
                V[e.v].d  = V[e.u].d+e.w;
                V[e.v].pi = e.u;
            }
        }
    }
}
```

Vergleichen von Positionsangaben

```
long long rund(ld x) {
    return (long long) round(x * pow(10, 3)); // auf 3 Nachkommastellen
}
bool gleich  (ld x, ld y) { return rund(x) == rund(y); }
bool groesser(ld x, ld y) { return rund(x) >  rund(y); }
bool kleiner (ld x, ld y) { return rund(x) <  rund(y); }
```

Prüfen nach Schneiden von Hindernissen

```
bool schneidetHindernis(ld x1, ld y1, ld x2, ld y2) {
    for(seite q : S)
        if(schneidetSeite(x1, y1, x2, y2, q))
            return true;
    return false;
}
bool schneidetSeite(ld x1, ld y1, ld x2, ld y2, seite q) {
    if(gleich(x1, x2) && gleich(q.x1, q.x2)) { // parallel
        return false;
    } else if(gleich(  x1,   x2)) {
        // Strecke senkrecht, Code aehnlich wie wenn nicht senkrecht
    } else if(gleich(q.x1, q.x2)) {
        // Seite   senkrecht, Code aehnlich wie wenn nicht senkrecht
    } else {
        // Geradengleichungen
        ld m_p = (  y2 -   y1) / (  x2 -   x1);
        ld m_q = (q.y2 - q.y1) / (q.x2 - q.x1);
        ld c_p =   y1 - m_p *   x1;
        ld c_q = q.y1 - m_q * q.x1;

        if(gleich(m_p, m_q)) // parallel
            return false; // da Fall 1

        ld xschnitt = (c_q-c_p)/(m_p-m_q);
        // Schnittpunkt muss auf beiden liegen
        if(groesser(xschnitt, max(  x1,   x2)) ||
           kleiner (xschnitt, min(  x1,   x2)) ||
           groesser(xschnitt, max(q.x1, q.x2)) ||
           kleiner (xschnitt, min(q.x1, q.x2)))
            return false; // da Fall 2.1

        if(!gleich(xschnitt, q.x1) && !gleich(xschnitt, q.x2))
```

```
            return true;  // da Fall 2.2

        bool imSinn;
        ld yschnitt, xq3, yq3, xq4, yq4;
        if (gleich(xschnitt, q.x1)) {
            xschnitt = q.x1;
            yschnitt = q.y1;
            xq3 = q.x2;
            yq3 = q.y2;
            xq4 = q.xan1;
            yq4 = q.yan1;
            imSinn = !q.imSinn;
        }
        if (gleich(xschnitt, q.x2)) {
            xschnitt = q.x2;
            yschnitt = q.y2;
            xq3 = q.x1;
            yq3 = q.y1;
            xq4 = q.xan2;
            yq4 = q.yan2;
            imSinn = q.imSinn;
        }
        ld term = (xq4-xq3)*(yschnitt-yq3) - (xschnitt-xq3)*(yq4-yq3);
        bool konvex = (term < 0 && !imSinn) || (term > 0 && imSinn);
        bool schneidet = schneidetStrecke(x1,  y1,  x2,  y2,
                                          xq3, yq3, xq4, yq4);
        bool imEck1 = imDreieck(xq3, yq3, xschnitt, yschnitt,
                                xq4, yq4, x1,       y1);
        bool imEck2 = imDreieck(xq3, yq3, xschnitt, yschnitt,
                                xq4, yq4, x2,       y2);
        if(konvex)
            return schneidet || imEck1 || imEck2;
        else {
            if(gleich(xschnitt, x1))
                return !(schneidet || imEck2);
            if(gleich(xschnitt, x2))
                return !(schneidet || imEck1);
            return true;
        }
    }
}
```

```
bool imDreieck(ld x1, ld y1, ld x2, ld y2, ld x3, ld y3, ld x, ld y) {
    ld A  = flaeche(x1, y1, x2, y2, x3, y3);
    ld A1 = flaeche(x,  y,  x2, y2, x3, y3);
    ld A2 = flaeche(x1, y1, x,  y,  x3, y3);
    ld A3 = flaeche(x1, y1, x2, y2, x,  y);
    return !gleich(A1, 0) && !gleich(A2, 0) && !gleich(A3, 0) &&
           gleich(A, A1+A2+A3);
}
ld flaeche(ld x1, ld y1, ld x2, ld y2, ld x3, ld y3) {
    return abs(x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2)) / 2;
}
```

```
bool schneidetStrecke(ld x1, ld y1, ld x2, ld y2,
                      ld x3, ld y3, ld x4, ld y4) {
    if(gleich(x1, x2) && gleich(x3, x4)) { // parallel
        return false;
    } else if(gleich(x1, x2)) { // Strecke senkrecht
        ld xschnitt = x1;
        ld m34 = (y4-y3)/(x4-x3);
        ld c34 =  y3 - m34*x3;
        ld yschnitt = m34*xschnitt + c34;
        // Schnittpunkt muss auf beiden liegen
        return kleiner (xschnitt, max(x3,x4)) &&
               groesser(xschnitt, min(x3,x4)) &&
               kleiner (yschnitt, max(y1,y2)) &&
               groesser(yschnitt, min(y1,y2));
    } else if(gleich(x3, x4)) { // Strecke senkrecht
        ld xschnitt = x3;
        ld m12 = (y2-y1)/(x2-x1);
        ld c12 =  y1 - m12*x1;
        ld yschnitt = m12*xschnitt + c12;
        // Schnittpunkt muss auf beiden liegen
        return kleiner (xschnitt, max(x1,x2)) &&
               groesser(xschnitt, min(x1,x2)) &&
               kleiner (yschnitt, max(y3,y4)) &&
               groesser(yschnitt, min(y3,y4));
    } else {
        ld m12 = (y2-y1)/(x2-x1);
        ld m34 = (y4-y3)/(x4-x3);
        ld c12 =  y1 - m12*x1;
        ld c34 =  y3 - m34*x3;
        if(gleich(m12, m34)) // parallel
            return false;
        else {
            ld xschnitt = (c34-c12)/(m12-m34);
            // Schnittpunkt muss auf beiden liegen
            return kleiner (xschnitt, max(x1,x2)) &&
                   groesser(xschnitt, min(x1,x2)) &&
                   kleiner (xschnitt, max(x3,x4)) &&
                   groesser(xschnitt, min(x3,x4));
        }
    }
}
```