Aufgabe 1
„Lisa rennt“- Dokumentation

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Lukas Rost

Teilnahme-ID: 48125

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

1.1 Das Geometric-Shortest-Path-Problem

Das der Aufgabe zugrundeliegende Problem nennt sich *Geometric Shortest Path* (GSP), auf Deutsch auch bekannt als Problem des geometrischen kürzesten Weges. Bei diesem Problem ist ein punktförmiger Roboter (oder auch eine Schülerin namens Lisa) gegeben, der/die sich an einer Startposition p_{start} in einem kartesischen Koordinatensystem befindet. In diesem Koordinatensystem befinden sich mehrere als Polygone modellierte Hindernisse, wobei jedes einzelne Polygon durch seine Eckpunkte gegeben ist.[1] Weiterhin ist eine Position p_{ziel} gegeben. Nun soll ein möglichst kurzer Weg von p_{start} zu p_{ziel} gefunden werden, wobei dieser nicht durch Hindernisse führen soll.[2][2]

Das hier gegebene Problem unterscheidet sich von GSP dadurch, dass keine Position p_{ziel}, sondern ein Strahl s_{ziel} (in Form eines beliebigen Punktes auf dem Strahl) erreicht werden soll. In diesem Fall handelt es sich dabei um die y-Achse ($x = 0$ mit $y \geq 0$). Zusätzlich soll nicht unbedingt der Weg optimiert werden, sondern die Startzeit, die abhängig von der Länge des Weges und der für jeden Punkt des Strahls unterschiedlichen Zielzeit ist. Diese Zielzeit kann mithilfe des Abstands des Punktes vom Ursprung berechnet werden.

Das Geometric-Shortest-Path-Problem wird im Allgemeinen in zwei Schritten gelöst: Zunächst wird ein Sichtbarkeitsgraph erstellt, auf welchem dann Dijkstras Algorithmus ausgeführt wird. Zur Lösung des hier gegebenen Problems wird jedoch ein zusätzlicher Schritt benötigt, der auf der Lösung des Problems ohne Hindernisse basiert. Diese drei Algorithmen sollen in den folgenden Abschnitten vorgestellt und näher erläutert werden.

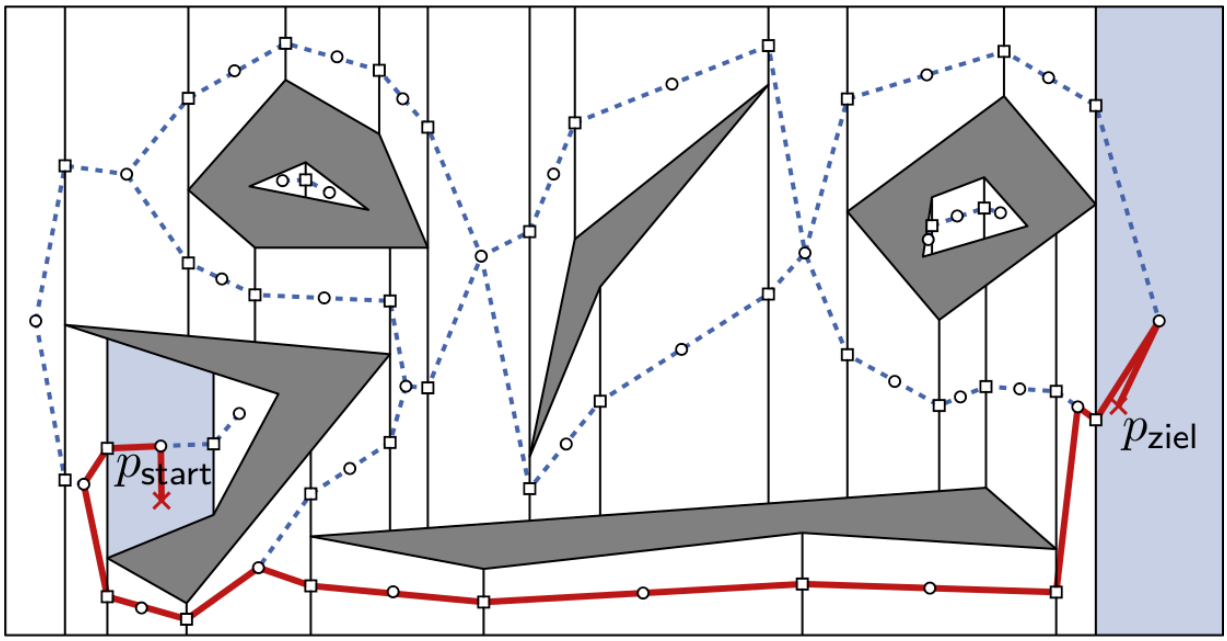


Abbildung 1: Illustration des GSP-Problems (aus [2])

[1] In dieser Dokumentation wird angenommen, dass die Punkte entgegen dem Uhrzeigersinn sortiert sind, sonst muss dies vorher geschehen.

[2] Zumindest nicht, wenn Lisa als Zielposition nicht das Krankenhaus erreichen will.

1.2 Erzeugung eines Sichtbarkeitsgraphen

Es lässt sich beobachten, dass der kürzeste s-t-Weg (dabei sei s die Startposition und t die Zielposition) in einem solchen Koordinatensystem ein Polygonzug sein muss, dessen innere Knoten Knoten (bzw. Ecken) der Hindernisse sind. Andernfalls wäre es immer möglich, einen kürzeren Weg zu konstruieren, der über einen Hindernisknoten führt.[2] Ausgehend davon lässt sich ein sogenannter Sichtbarkeitsgraph erzeugen, bei dem es sich um einen gewichteten, ungerichteten Graphen handelt.

Für eine Polygonmenge S mit Knotenmenge $V(S)$ sei dieser definiert als $G_{vis}(S) = (V(S), E_{vis}(S))$. Dabei ist $E_{vis}(S)$ die Menge der Knotenpaare von S, sodass die dazwischenliegende Strecke kein Polygon (bzw. das Innere eines Polygons) schneidet. Mathematisch ausgedrückt bedeutet das $E_{vis}(S) = \{uv | u, v \in V(S) \text{ und } \overline{uv} \subset C_{free} = \mathbb{R}^2 \setminus \bigcup S\}$. Das Gewicht einer Kante sei dabei als euklidischer Abstand der beiden Endpunkte definiert.

Wenn man S^* als $S \cup \{s\}$ definiert (bei normalen Sichtbarkeitsgraphen wird auch t hinzugefügt) und den Sichtbarkeitsgraphen dafür analog, kann man damit das GSP-Problem lösen. Nun entspricht der kürzeste s-t-Weg, der Hindernisse vermeidet, einem kürzesten Weg in $G_{vis}(S^*)$.

1.2.1 Naiver Ansatz

$G_{vis}(S^*)$ lässt sich nun naiv in $\mathcal{O}(n^3)$ berechnen, wobei $n = |V(S^*)|$. Dazu bestimmt man für jeden Knoten $u \in V(S*)$ die von ihm sichtbaren Knoten v. Dabei muss man für jede Strecke $\overline{uv}$ prüfen, ob sie eine der $|E_{vis}(S^*)| = \mathcal{O}(n)$ in Frage kommenden Hinderniskanten schneidet. Die Bestimmung der von u sichtbaren Knoten ist somit in $\mathcal{O}(n^2)$ durchführbar und insgesamt ergibt sich eine Laufzeit von $O(n^3)$.

Die Funktionsweise des naiven Algorithmus wird auch in folgendem Pseudocode deutlich:

Algorithmus 1 Naiver Algorithmus

```
function visibilityGraph(S)
    G = (V(S), E) ← leerer Sichtbarkeitsgraph
    for all Knoten v ∈ V(S) do                          ▷ O(n)
        for all Knoten w ∈ V(S) \ {v} do                ▷ O(n)
            for all Kante e ∈ E_vis(S) do               ▷ O(n)
                if vw schneidet keine der Kanten e then
                    E ← E ∪ {vw}
                end if
            end for
        end for
    end for
    return G
end function
```

1.2.2 Der Lee-Algorithmus

Es ist jedoch auch möglich, die Bestimmung der von u sichtbaren Knoten in $\mathcal{O}(n \cdot \log n)$ durchzuführen, wodurch sich insgesamt eine Laufzeit von $\mathcal{O}(n^2 \cdot \log n)$ ergibt. Der dazu

notwendige Algorithmus ist der Algorithmus von D. T. Lee. Durch dessen geringere Laufzeit lässt sich insbesondere bei großen Eingaben eine deutliche Verbesserung erreichen.

Es existieren zwar noch schnellere Algorithmen wie der nach Overmars und Welzl in $\mathcal{O}(n^2)$ oder der nach Ghosh und Mount in $\mathcal{O}(n \cdot \log n + E)$[5]. Doch die damit erreichten Verbesserungen sind nur in speziellen Fällen wirklich bemerkbar, während die Implementierung deutlich schwieriger ist.

Lees Algorithmus ist grundlegend ähnlich aufgebaut, muss jedoch für jede Strecke $\overline{uv}$ nur noch eine Hinderniskante prüfen, welche in $\mathcal{O}(\log n)$ bestimmt werden kann. Dies wird auch in folgendem Pseudocode deutlich:

Algorithmus 2 Der Algorithmus von Lee

```
function visibilityGraph(S)
    G = (V(S), E) ← leerer Sichtbarkeitsgraph
    for all Knoten v ∈ V(S) do                          ▷ O(n)
        W ← visible_vertices(v,S)                       ▷ O(n · log n)
        E ← E ∪ {vw | w ∈ W}
    end for
end function
```

Der Algorithmus in der Methode `visible_vertices` benutzt dabei eine sogenannte rotierende *Sweep line* beziehungsweise *Scan line*. Diese Sweep line fegt („sweept“) den zweidimensionalen Raum aus, wobei sie durch den gesamten Raum bewegt wird, bis alle Objekte (in diesem Fall die Knoten) besucht und verarbeitet wurden. Im Falle von Lees Algorithmus rotiert die Sweep line (mathematisch gesehen ein Strahl) um dem Startpunkt v gegen den Uhrzeigersinn.

Am Anfang zeigt die Sweep line nach rechts (parallel zur x-Achse) und rotiert so lange gegen den Uhrzeigersinn, bis sie einen Punkt trifft, der auf Sichtbarkeit überprüft werden muss. Dazu wird eine Liste der offenen Kanten geführt, mit denen sich die Sweep line aktuell überhaupt schneiden kann und die damit relevant sind.[4] Anfangs werden dabei einmal alle Kanten daraufhin überprüft, ob sie sich mit der Sweep line (in der Anfangsstellung) schneiden. Entsprechend wird dann diese Liste initialisiert.

Trifft die Sweep line auf einen Punkt a, werden deshalb alle Kanten überprüft, deren Endpunkt a ist. Liegt eine Kante bezüglich der Sweep line im Uhrzeigersinn (*clockwise side*), kann sie, sofern sie enthalten ist, aus der Liste der offenen Kanten entfernt werden, da die Sweep line bei Bewegung gegen den Uhrzeigersinn nie wieder mit ihr in Berührung kommt. Liegt die Kante dagegen entgegen des Uhrzeigersinns (*counter-clockwise side*), so muss sie zu den offenen Kanten hinzugefügt werden, da sich die Sweep line im Folgenden mit ihr schneiden kann.

So werden in diesem Beispiel (siehe nächste Seite) am Punkt a die Kanten $\overline{ab}$ und $\overline{ac}$ hinzugefügt, da sie bezüglich der Sweep line gegen den Uhrzeigersinn (auf ihrer linken Seite) liegen. In Punkt b kann $\overline{ab}$ dagegen wieder entfernt werden, da sie rechts der Sweep line liegt. Hier wird dann jedoch $\overline{bc}$ hinzugefügt.

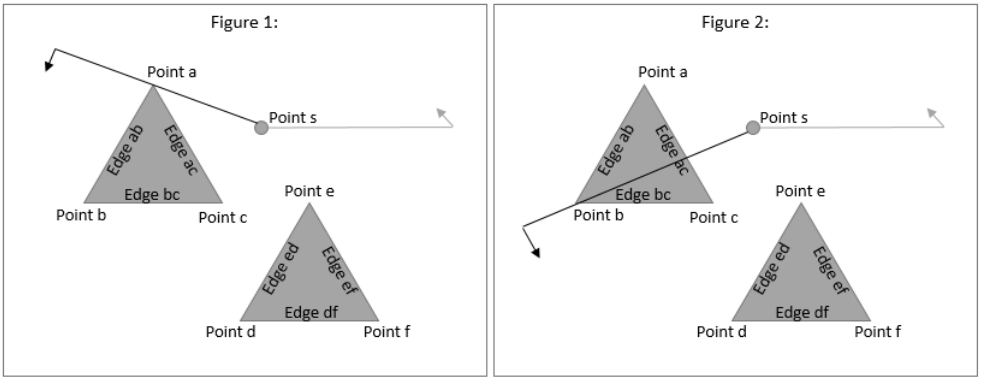


Abbildung 2: Illustration des Konzepts der *Sweep line* (aus [4])

Es lässt sich zeigen, dass sogar nur diejenige offene Kante geprüft werden muss, welche am nächsten an v liegt, d.h. diejenige, für die der Schnittpunkt mit der Sweep line am nächsten an v liegt. Um diese Kante schnell zu bestimmen, muss man die Kanten nach der Distanz zum Schnittpunkt ordnen. Dies lässt sich zum Beispiel mit einem balancierten binären Suchbaum wie einem *AVL-Baum* erreichen.[3][4]

Die `visible_vertices`-Funktion sieht nun also bei einer Formulierung als Pseudocode ungefähr so aus:

Algorithmus 3 Bestimmung der sichtbaren Knoten bzw. Punkte im Lee-Algorithmus

function VISIBLE_VERTICES(v, S)
  $w \leftarrow sort(V(S) \setminus \{v\})$ ▷ Sortieren der Knoten (siehe unten)
  $s \leftarrow v + \lambda \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ mit $\lambda \in \mathbb{R}^+$ ▷ Sweep line initialisieren
  $T \leftarrow binarySearchTree()$
  $W \leftarrow \emptyset$
  for all Kante $e \in E(S)$ **do**
    if $e \cap s \neq \emptyset$ **then** ▷ alle Kanten, die die Sweep line schneiden
      $T \leftarrow T \cup e$
    end if
  end for
  for all Knoten $w_i \in w$ **do** ▷ in sortierter Reihenfolge
    Rotiere s so, dass er w_i schneidet
    if VISIBLE(w_i) **then** ▷ ist sichtbar
      $W \leftarrow W \cup w_i$
    end if
    for all Kante e mit Endpunkt w_i **do**
      if e links von s **then**
        $T \leftarrow T \cup \{e\}$ ▷ Kante hinzufügen
      end if
      if e rechts von s **then**
        $T \leftarrow T \setminus \{e\}$ ▷ Kante entfernen
      end if
    end for
  end for
  return W
end function

[3]Die Funktionsweise eines AVL-Baums sei hier als bekannt vorausgesetzt.

Das Sortieren der Knoten erfolgt dabei anhand ihrer Polarkoordinaten relativ zu v. Dabei wird zuerst nach kleinerem Winkel zu s und dann, falls dieser gleich sein sollte nach kleinerem Abstand zu v sortiert. Die Formulierung *links* im Pseudocode bedeutet entgegengesetzt dem Uhrzeigersinn und *rechts* bedeutet im Uhrzeigersinn.

Algorithmus 4 Sichtbarkeitsprüfung im Lee-Algorithmus

```
function VISIBLE(w_i)
    if T = ∅ then
        return true
    end if
    if overline(vw_i) ∩ smallest(T) = ∅ then
        return true
    else
        return false
    end if
end function
```

Diese Funktion `visible` prüft dabei ganz einfach die Sichtbarkeit eines Knotens. $smallest(T)$ gibt die kleinste Kante im Suchbaum an.

1.3 Der Dijkstra-Algorithmus

Um nun im Sichtbarkeitsgraphen einen kürzesten Pfad vom Startknoten s zu allen anderen Knoten zu bestimmen, was für die Lösung des Problems notwendig ist, kann man Dijkstras Algorithmus verwenden. Dieser arbeitet nach dem Greedy-Prinzip, bei dem in jedem Schritt ein optimaler Folgezustand gewählt wird, der das aktuell beste Ergebnis verspricht. Er arbeitet grundlegend wie folgt:

1. Weise allen Knoten die beiden Eigenschaften „Distanz“ und „Vorgänger“ zu. Initialisiere die Distanz im Startknoten s mit 0 und in allen anderen Knoten mit ∞.
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
   a) speichere, dass dieser Knoten schon besucht wurde.
   b) Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten.
   c) Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.

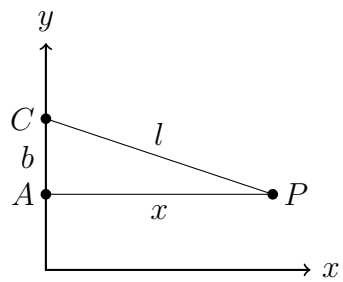
Der Dijkstra-Algorithmus hat eine Laufzeit von $\mathcal{O}((|V| + |E|) \cdot \log |V|)$ (bei Implementierung mit einer Vorrangwarteschlange). Da der Algorithmus den entsprechenden Weg durch das Setzen eines Vorgängers ebenfalls bestimmt, kann auch dieser selbst ausgegeben werden.

1.4 Lösung des Problems ohne Hindernisse

Damit man nun diese spezielle Abwandlung des Problems lösen kann, sollte man zunächst einmal versuchen, das Problem ohne Hindernisse zu lösen, wie dies die Aufgabenstellung auch *sehr* unauffällig nahelegt.

Zunächst lässt sich feststellen, dass es ausschließlich sinnvoll ist, Punkte mit einer y-Koordinate $\geq$ der y-Koordinate von Lisas Haus als Zielpunkte zu wählen. Sonst würde der Abstand zwischen Start- und Zielpunkt größer, während die Abfahrtszeit des Busses früher läge, was insgesamt für Lisa ein früheres Aufstehen bedeuten würde.

Betrachten wir nun die nebenstehende Situation, in der $P(x_p|y_p)$ Lisas Haus darstellt und $C(x_c|y_c)$ den optimalen Zielpunkt mit $y_c \geq y_p$. Die Zeit, die der Bus für die Strecke b benötigt, berechnet sich mit v_b als Geschwindigkeit des Busses zu



$$t_B = \frac{b}{v_B} \tag{1}$$

Die Zeit, welche Lisa für die Strecke l benötigt, berechnet sich äquivalent zu

$$t_L = \frac{l}{v_L} = \frac{\sqrt{x^2+b^2}}{v_L} \tag{2}$$

da das Dreieck rechtwinklig ist. Um nun den optimalen Punkt C zu finden, ist es nötig, den Term $t_L - t_B$ abhängig von der Strecke b zu minimieren. Dies lässt sich einfach mit dem aus der Kurvendiskussion bekannten Ansatz über die erste Ableitung erreichen. Dann ergibt sich:

$$\begin{aligned}\frac{d(t_L - t_B)}{db} &= \frac{d}{db}\left(\frac{\sqrt{x^2+b^2}}{v_L} - \frac{b}{v_B}\right) \\ &= \frac{1}{v_L} \cdot \frac{2 \cdot b}{2 \cdot \sqrt{x^2+b^2}} - \frac{1}{v_B} \\ &= \frac{1}{v_L} \cdot \frac{b}{\sqrt{x^2+b^2}} - \frac{1}{v_B}\end{aligned} \tag{3}$$

Setzt man diese Ableitung nun gleich 0, erhält man:

$$\frac{1}{v_L} \cdot \frac{b}{\sqrt{x^2+b^2}} - \frac{1}{v_B} = 0 \tag{4}$$

$$\frac{1}{v_L} \cdot \frac{b}{\sqrt{x^2+b^2}} = \frac{1}{v_B} \tag{5}$$

$$b = \frac{v_L}{v_B} \cdot \sqrt{x^2+b^2} \tag{6}$$

$$b^2 = \frac{v_L^2}{v_B^2} \cdot (x^2+b^2) \tag{7}$$

$$(1 - \frac{v_L^2}{v_B^2}) \cdot b^2 = \frac{v_L^2}{v_B^2} \cdot x^2 \tag{8}$$

$$b^2 = \frac{v_L^2}{v_B^2} \cdot \frac{1}{(1 - \frac{v_L^2}{v_B^2})} \cdot x^2 \tag{9}$$

$$b = \frac{v_L}{v_B} \cdot \frac{1}{\sqrt{1 - \frac{v_L^2}{v_B^2}}} \cdot x \tag{10}$$

Für den optimalen Zielpunkt C gilt somit $x_c = 0$ und $y_c = y_p + \frac{v_L}{v_B} \cdot \frac{1}{\sqrt{1-\frac{v_L^2}{v_B^2}}} \cdot x_p$. [4] Setzt man die in der Aufgabenstellung gegebenen Geschwindigkeiten ein. ergibt sich: $y_c = y_p + \frac{\sqrt{3}}{3} \cdot x_p$

[4] Ich habe keine Ahnung, warum das wie der Lorentzfaktor aus der Relativitätstheorie aussieht.

1.5 Kombination der Ansätze

Um das Problem nun lösen zu können, muss man beide Ansätze (mit und ohne Hindernisse) kombinieren. Dazu bestimmt man zu jedem Knoten/Punkt p in der Eingabe einen (von mir so bezeichneten) *Companion*-Punkt c, indem man die im vorigen Abschnitt berechnete Gleichung auf diesen Punkt anwendet. Dieser Punkt c ist dann der optimale Punkt, um von p aus die y-Achse zu erreichen und dabei möglichst spät starten zu müssen.

Da es jedoch sein kann, dass c von p nicht sichtbar ist (ein Hindernis liegt dazwischen), muss man eine Sichtbarkeitsprüfung durchführen. Dazu betrachtet man c einfach bei der Erstellung des Sichtbarkeitsgraphen, genauer bei der Ermittlung der sichtbaren Punkte von p aus (und nur dabei, denn für alle anderen Punkte hat c keine Relevanz).

Nun kann man mithilfe des Dijkstra-Algorithmus' die kürzesten Wege zu allen Companion-Knoten berechnen. Aus diesen und der Lage der Companion-Punkte (aus dieser bestimmt sich die Zeit, an der der Bus dort vorbeifährt) lässt sich dann für jeden Companion-Punkt die spätestmögliche Startzeit am Startpunkt s bestimmen, wenn man an diesem Companion-Punkt auf den Bus treffen will.

Der Companion-Punkt mit der spätesten spätestmöglichen Startzeit ist der Punkt, an dem Lisa auf den Bus treffen sollte, und der Weg zu ihm der optimale Weg.

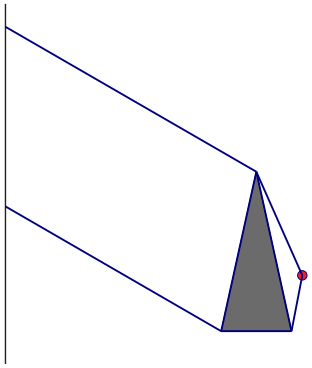


Abbildung 3: Veranschaulichung eines kompletten Sichtbarkeitsgraphen mit Companion-Punkten für Beispiel 1 des BwInf (blaue Linien sind Kanten des Graphen)

Hierbei soll noch kurz informell bewiesen werden, warum von einem Punkt p aus nur der Punkt c optimal sein kann und, wenn keine Sichtbarkeit zwischen diesen beiden Punkten besteht, c einfach ignoriert werden kann. Nehmen wir an, von p aus sei c nicht sichtbar, d.h. der Strahl von p in Richtung c führt durch ein Polygon. Nun können wir den Strahl so nach unten (gegen den Uhrzeigersinn) rotieren, dass er gerade kein Polygon mehr schneidet.

Nun schneidet er jedoch in jedem Fall einen Eckpunkt eines Polygons, hier d genannt. Die Strecke zwischen p und d muss im Sichtbarkeitsgraphen enthalten sein, muss also hier nicht erneut betrachtet werden. Von d aus gesehen gibt es einen eigenen Companion-Punkt c_2, der optimal ist. Also muss auch die Teilstrecke zwischen d und dem neuen Schnittpunkt des Strahls mit der y-Achse nicht beachtet werden.

1.6 Laufzeitbetrachtung

In den einzelnen Abschnitten wurde teilweise schon auf die Laufzeiten der Teilalgorithmen eingegangen. Hier soll dies noch einmal zusammengefasst werden.

Algorithmus von Lee. Die Funktion `visible_vertices` im Algorithmus von Lee läuft offensichtlich in $\mathcal{O}(n \cdot \log n)$. Der Grund dafür ist, dass die einzelnen Schritte folgende Laufzeiten haben:

- **Sortieren der Knoten**: kann in $\mathcal{O}(n \cdot \log n)$ erfolgen (mit z.B. Quicksort), wobei $\Omega(n \cdot \log n)$ wie allgemein bekannt auch die Untergrenze für die Worst-Case-Laufzeit eines Sortieralgorithmus ist.
- **Schnittprüfung von Sweep line und Kanten**: Diese muss anfangs für n Polygonkanten durchgeführt werden, hier ergibt sich also $\mathcal{O}(n)$.
- **Überprüfen aller Knoten**: Dies wird für alle $\mathcal{O}(n)$ Knoten durchgeführt. Dabei benötigt die Sichtbarkeitsprüfung $\mathcal{O}(\log n)$ (durch die Verwendung eines Suchbaumes). Danach müssen noch konstant viele Kanten dem Baum hinzugefügt bzw. entfernt werden[5] Das Hinzufügen bzw. Entfernen benötigt wieder $\mathcal{O}(\log n)$. Insgesamt ergibt sich also auch hier wieder $\mathcal{O}(n \cdot \log n)$.

Wie eindeutig sichtbar ist, dominiert hier $\mathcal{O}(n \cdot \log n)$. Da die Funktion für alle n Knoten aufgerufen wird, ergibt sich für den Gesamtalgorithmus eine Laufzeit in $\mathcal{O}(n^2 \cdot \log n)$ □

Algorithmus von Dijkstra. Der Dijkstra-Algorithmus hat bei der allgemein übliche Implementierung mit einer Priority Queue eine Laufzeit von $\mathcal{O}((|V|+|E|) \cdot \log |V|)$, wobei V die Knotenmenge und E die Kantenmenge ist. Da in diesem Fall $|V| \in \mathcal{O}(n)$ und $|E| \in \mathcal{O}(n^2)$ gilt, ergibt sich hier eine Laufzeit in $\mathcal{O}(n^2 \cdot \log n)$. □

Kombination der Ansätze. Da bei der Berechnung des Companion-Punktes nur einfache Grundrechenoperationen durchgeführt werden, kann dieser in konstanter Zeit, also $\mathcal{O}(1)$ berechnet werden. Für die n Punkte bzw. Knoten ergibt sich dann $\mathcal{O}(n)$. Alle weiteren Schritte aus dem Abschnitt *Kombination der Ansätze* sind in die Algorithmen von Dijkstra bzw. Lee integriert, tragen also nicht weiter zur Laufzeit bei. Nur die Berechnung der Startzeit muss für jeden Punkt noch separat erfolgen, hier ergibt sich wieder $\mathcal{O}(n)$. □

Insgesamt ergibt sich für die Laufzeit aller drei Algorithmen also $\mathcal{O}(n^2 \cdot \log n + n^2 \cdot \log n + n) = \mathcal{O}(n^2 \cdot \log n)$.

1.7 Erweiterungen

1.7.1 Anpassbare Geschwindigkeiten

Es ist relativ einfach möglich, die Geschwindigkeiten, mit denen sich Lisa und der Bus bewegen, zu verändern. Dazu muss einfach in allen Gleichungen, die aus Weg und Geschwindigkeit die benötigte Zeit berechnen, die Geschwindigkeit entsprechend gewählt werden. Da bei der *Lösung des Problems ohne Hindernisse* bereits die Geschwindigkeiten als Parameter verwendet wurden, ist dies auch dort ohne Probleme möglich. Diese Erweiterung läuft in $\mathcal{O}(1)$ und wurde **implementiert** (siehe Umsetzung).

1.7.2 Polygonale Lisa

Nun modellieren wir Lisa nicht mehr als Punkt, sondern als Polygon. Wieso? Möglicherweise handelt es sich bei einem der Hindernispolygone um das örtliche Fastfoodrestaurant und Lisa besucht dieses sehr oft, sodass sie extrem zugenommen hat.

[5] Normalerweise höchstens 2, in Sonderfällen können es aber auch bis zu 4 sein. Diese Sonderfälle treten dann auf, wenn sich zwei Polygone eine Kante oder einen Knoten teilen.

Möglicherweise ist der Grund aber auch unspektakulärer. Eventuell ist Lisa gerade 18 geworden und möchte nun mit ihrem Auto zur Bushaltestelle fahren, auch wenn man über die ökologische Sinnhaftigkeit dieses Vorhabens streiten kann.[6]

Dies lässt sich mathematisch beschreiben, indem man für jedes Hindernispolygon dessen Minkowski-Summe mit dem Lisa-Polygon berechnet und auf den so entstandenen Polygonen den ursprünglichen Algorithmus ausführt. Die Minkowski-Summe ist dabei für zwei Teilmengen A und B (in diesem Fall Polygone) eines Vektorraums (des $\mathbb{R}^2$) definiert als:

$$A + B := \{a + b \,|\, a \in A, b \in B\} \tag{11}$$

Sie ist also die Menge aller Elemente, die Summen von je einem Element aus A und einem aus B sind.[9] Dies entspricht im Prinzip dem Entlangfahren des Lisa-Polygons am Rand des Hindernispolygons. Dabei wird das Hindernis so verbreitert, dass Lisa nun wieder als punktförmig angenommen und der ursprüngliche Algorithmus ausgeführt werden kann.

Seien nun das Hindernispolygon $P = (v_1, ..., v_k)$ und das Lisa-Polygon $L = (w_1, ..., w_m)$ mit v_i und w_i als Eckpunkten[7], wobei v_1 und w_1 die Punkte mit der minimalen y-Koordinate[8] sind. Für diese zwei konvexen Polygone (im weiteren werden für diese Erweiterung nur solche betrachtet) im $\mathbb{R}^2$ kann die Minkowski-Summe wie folgt berechnet werden:

Algorithmus 5 Bestimmung der Minkowski-Summe

```
function minkowskiSum(P,L)
    i ← 1; j ← 1
    v_{k+1} = v_1; v_{k+2} = v_2; w_{m+1} = w_1; w_{m+2} = w_2
    S = ∅                                              ▷ leeres Polygon
    repeat
        S = S ∪ {v_i + w_j}
        if winkel(v_i, v_{i+1}) < winkel(w_j, w_{j+1}) then
            i ← i + 1
        else if winkel(v_i, v_{i+1}) > winkel(w_j, w_{j+1}) then
            j ← j + 1
        else
            i ← i + 1; j ← j + 1
        end if
    until i = k + 1 and j = m + 1
end function
```

Dabei gibt die Funktion $winkel(p, q)$ den Winkel zwischen der positiven x-Achse und dem Vektor pq zurück.

Dieser Algorithmus läuft offensichtlich in $\mathcal{O}(k+m)$, denn i wird höchstens k Mal inkrementiert und j m Mal. Da er $n_{Polygon}$ Mal[9] ausgeführt wird, erhält man $\mathcal{O}(\sum_1^{n_{Polygon}}(k_i+m_i)) = \mathcal{O}(n + n_{Polygon} \cdot m)$, wobei n die Anzahl aller Eckpunkte aller Polygone ist. Diese Erweiterung wurde **implementiert**.

[6] Hier Fridays-for-Future-Demonstration einfügen.
[7] gegen den Uhrzeigersinn geordnet
[8] Falls dies mehrere sind, derjenige mit der minimalen x-Koordinate.
[9] Anzahl der Polygone

1.7.3 Nicht-polygonale Hindernisse

Es wäre auch möglich, die Art der Hindernisse auf Flächen, die keine Polygone sind, zu erweitern. Dies könnten beispielsweise Ovale oder Kreise sein. Eine Möglichkeit, den Sichtbarkeitsgraph darauf zu erweitern, ist in [8] beschrieben. Dabei muss man anstelle des klassischen Sichtbarkeitsgraphen einen Tangenten-Sichtbarkeitsgraphen verwenden. Da dies jedoch relativ kompliziert ist, wird an dieser Stelle nicht genauer darauf eingegangen.

Diese Erweiterung wurde **nicht implementiert**.

1.7.4 Das Problem im $\mathbb{R}^3$

Nehmen wir nun an, dass Lisa fliegen kann. Wie sie diese Fähigkeit erlangt hat, ist unwichtig. Möglicherweise ist sie mit Quax aus der letztjährigen 3. Aufgabe der 2. Runde befreundet und kann seinen Quadrocopter benutzen. Außerdem nehmen wir an, dass der Bus unendlich hoch ist. Inwiefern das sinnvoll ist, soll hier ebenfalls nicht betrachtet werden.

Dazu kann man das Koordinatensystem von einem $\mathbb{R}^2$ auf einen $\mathbb{R}^3$ erweitern, sodass ein dreidimensionales Koordinatensystem entsteht. In diesem sind die Hindernisse dann Körper und die Zielgerade eine Zielebene, genauer die yz-Ebene. Ein Ansatz dazu ist in [6] beschrieben.

Ein solcher Sichtbarkeitsgraph ist dann jedoch ein Hypergraph, bei dem jede Kante 3 Knoten verbindet. Aus diesem Grund wurde die Erweiterung **nicht implementiert**.

Literatur

[1] Bittel, O. (HTWG Konstanz): Autonome Roboter - Wegekartenverfahren, SS 2016 (Präsentation), `http://www-home.htwg-konstanz.de/~bittel/msi_robo/Vorlesung/06_Planung_Wegekarten.pdf`

[2] Nöllenburg, Martin (KIT): Vorlesung Algorithmische Geometrie - Sichtbarkeitsgraphen, 2011 (Präsentation), `https://i11www.iti.kit.edu/_media/teaching/sommer2011/compgeom/algogeom-ss11-vl14-printer.pdf`

[3] Reksten-Monsen, Christian: Distance Tables Part 1 - Defining the Problem, `https://taipanrex.github.io/2016/09/17/Distance-Tables-Part-1-Defining-the-Problem.html`

[4] Reksten-Monsen, Christian: Distance Tables Part 2 - Lee's Visibility Graph Algorithm, `https://taipanrex.github.io/2016/10/19/Distance-Tables-Part-2-Lees-Visibility-Graph-Algorithm.html`

[5] Kitzinger, John (University of New Mexico): The Visibility Graph Among Polygonal Obstacles: A Comparison of Algorithms, 2003, `https://www.cs.unm.edu/~moore/tr/03-05/Kitzingerthesis.pdf`

[6] Bygi, Mojtaba Nouri; Ghodsi, Mohammad (Sharif University of Technology): 3D Visibility Graph, `https://pdfs.semanticscholar.org/aba1/5853197c24ed6f164e4fb5e2f134462c7ebf.pdf`

[7] Coleman, Dave (University of Colorado): Lee's Visibility Graph Algorithm - Implementation and Analysis, 2012, https://github.com/davetcoleman/visibility_graph/blob/master/Visibility_Graph_Algorithm.pdf

[8] Hutchinson, Joan P. (Macalester College): Arc- and circle-visibility graphs, https://pdfs.semanticscholar.org/d257/d8f5ea2f9bb32c555b4d5723fdcf1e97dc4f.pdf

[9] Skript der Uni Freiburg zur Minkowski-Summe, http://algo.informatik.uni-freiburg.de/bibliothek/books/ad-buch/k7/slides/08.pdf

2 Umsetzung

2.1 Allgemeine Hinweise zur Benutzung

Das Programm wurde in der Programmiersprache Python 3.7 implementiert. Zur Verwaltung und Erstellung des Sichtbarkeitsgraphen wurde die Bibliothek `pyvisgraph` von Christian Reksten-Monsen benutzt. Dabei kommt jedoch eine von mir erheblich veränderte Version zum Einsatz, die an das hier zu lösende Problem angepasst ist. Diese Version ist die einzige mit dem Programm kompatible und ist in die Einsendung integriert.

Diese Bibliothek benötigt wiederum die Bibliothek tqdm (`sudo pip3 install tqdm`). Alle weiteren verwendeten Bibliotheken sind üblicherweise vorinstalliert.

Die Eingabe und Ausgabe des Programms erfolgt in Dateien, die mithilfe der Konsolenparameter frei gewählt werden können. Die weitere Bedienung sollte selbsterklärend sein. Es gibt folgende Konsolenparameter:

```
usage: main.py [-h] [-i INPUT] [-o OUTPUT] [-so SVG] [-d]
[-vlisa VELOCITY_LISA] [-vbus VELOCITY_BUS]
[-minkowski MINKOWSKI]

Lösung zu Lisa rennt, Aufgabe 1, Runde 2, 37. BwInf von Lukas Rost

optional arguments:
-h, --help            show this help message and exit
-i INPUT              Eingabedatei
-o OUTPUT             Ausgabedatei
-so SVG               SVG-Ausgabedatei
-d                    Debug-Ausgaben aktivieren
-vlisa VELOCITY_LISA  Erweiterung Geschwindigkeiten: Lisa in km/h
-vbus VELOCITY_BUS    Erweiterung Geschwindigkeiten: Bus in km/h
-minkowski MINKOWSKI  Erweiterung Minkowski-Summe: Eingabedatei (1 Polygon
im gleichen Format wie in der normalen Eingabe)
```

Die Eingabedatei für die Minkowski-Erweiterung ist im Format der anderen Eingaben gestaltet. Durch Leerzeichen getrennt wird zunächst die Anzahl der Eckpunkte und anschließend für jeden Eckpunkt x- und y-Koordinate angegeben. Dabei stehen alle Eingaben in einer Zeile (Beispiel: `4 0 0 0 1 1 1 1 0`).

2.2 Struktur des Programms und Implementierung der Algorithmen

3 Beispiele

3.1 Beispiel 1

3.2 Beispiel 2

3.3 Beispiel 3

3.4 Beispiel 4

3.5 Beispiel 5

3.6 Eigene Beispiele

3.7 Beispiele für die Erweiterungen

4 Quellcode

```
1  #Main program
2  import argparse
3  import pyvisgraph as vg
4
5  #Commandline-Argumente parsen
6  parser = argparse.ArgumentParser(description="Lösung zu Lisa rennt, Aufgabe 1,
   ↪  Runde 2, 37. BwInf von Lukas Rost")
7
8  parser.add_argument('-i',
   ↪  action="store",dest="input",default="lisarennt1.txt",help="Eingabedatei")
9  parser.add_argument('-o',action="store",dest="output",⌋
   ↪  default="lisarennt1_output.txt",help="Ausgabedatei")
10 parser.add_argument('-so', action="store",dest="svg",⌋
   ↪  default="lisarennt1_svg.svg",help="SVG-Ausgabedatei")
11 parser.add_argument('-d',action="store_true",default=False,dest="debug",⌋
   ↪  help="Debug-Ausgaben
   ↪  aktivieren")
12 parser.add_argument('-vlisa',action="store",dest="velocity_lisa",default=15,⌋
   ↪  type=float,help="Erweiterung Geschwindigkeiten: Lisa in
   ↪  km/h")
13 parser.add_argument('-vbus',action="store",dest="velocity_bus",default=30,⌋
   ↪  type=float,help="Erweiterung Geschwindigkeiten: Bus in
   ↪  km/h")
14 parser.add_argument('-minkowski',action="store",default=None,help="Erweiterung
   ↪  Minkowski-Summe: Eingabedatei (1 Polygon im gleichen Format wie in der
   ↪  normalen Eingabe)")
15
16 args = parser.parse_args()
17
18 # Polygone einlesen
19 infile = open(args.input,'r')
20 numpoly = int(infile.readline())
21 polylist = []
22
23 for i in range(numpoly):
```

```
24      pointlist = []
25      line = infile.readline().split(" ")
26      index = 1
27      for j in range(line[0]):
28          pointlist.append(vg.Point(line[index],line(index+1),
            ↪  polygon_id="P"+(j+1)))
29          index += 2
30      polylist.append(pointlist)
31
32  #Lisas Position einlesen
33  pos = infile.readline.split(" ")
34  infile.close()
35
36  #Graph erstellen
37  graph = vg.VisGraph()
38  graph.build(polylist)
39  shortest = graph.shortest_path(vg.Point(pos[0],pos[i],polygon_id="L")) #TODO
    ↪  change visgraph library
40
41  #Debug-Ausgaben
42  graph.save(args.input + "_pickle.pk1")
43
44  #Ausgabe
45  #TODO
46  #TODO Zeitmessungen
47
48  #TODO Erweiterungen
```

Quellcode 1: Auszug aus dem Hauptprogramm (*main.py*), soweit möglich ohne Eingabe und Ausgabe