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.

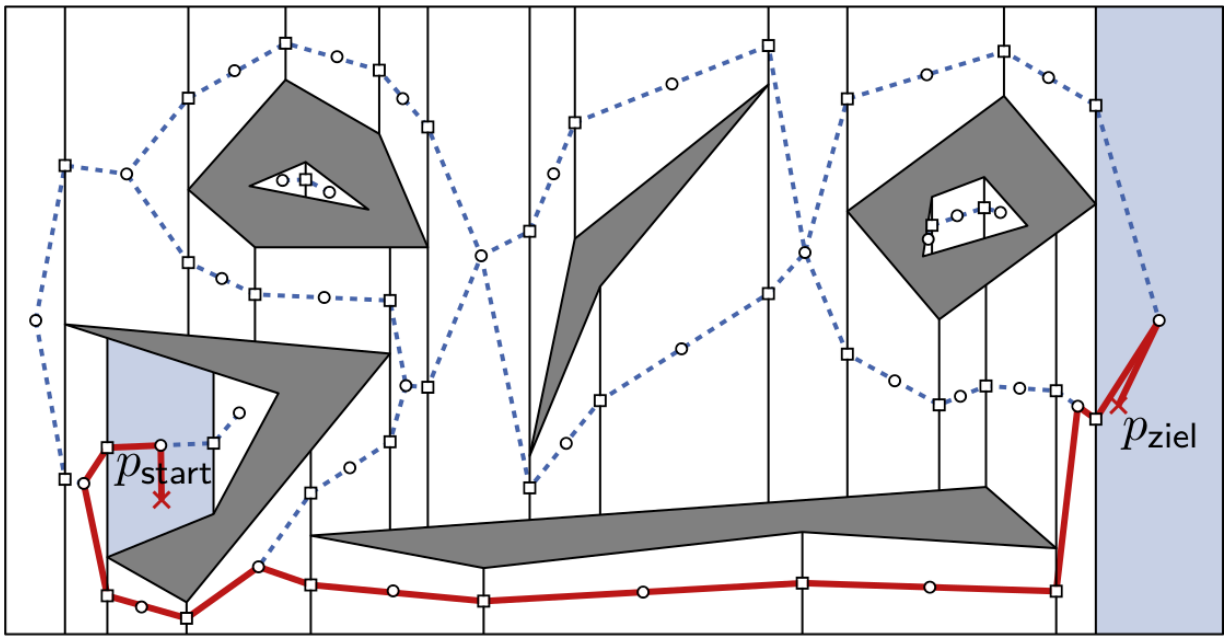

Abbildung 1: Illustration des GSP-Problems (aus [2])

[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.

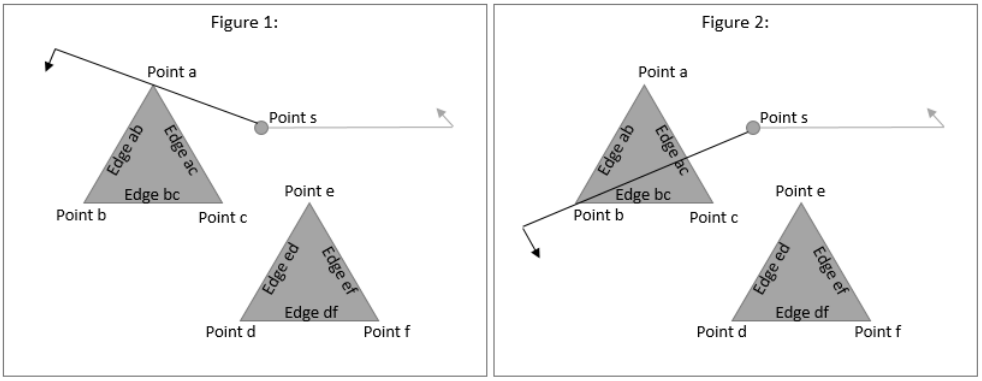

Abbildung 2: Illustration des Konzepts der *Sweep line* (aus [4])

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.[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

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 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

1.5 Kombination der Ansätze

1.6 Laufzeitbetrachtung

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 Andere Zielgeraden

Auch die Gerade bzw. der Strahl, auf der bzw. dem der Bus fährt, kann geändert werden. Dazu muss einfach das Koordinatensystem so transformiert werden, dass die Gerade im ursprünglichen Koordinatensystem der y-Achse im transformierten Koordinatensystem entspricht. Dies kann durch Drehungen und Parallelverschiebungen geschehen.

Die Erweiterung läuft pro Punkt in konstanter Laufzeit, also insgesamt in $\mathcal{O}(n)$. Diese Erweiterung wurde **implementiert**.

1.7.3 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.

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.[3]

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:

$$A + B := \{a + b \,|\, a \in A, b \in B\} \tag{1}$$

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 $P = (v_1, ..., v_n)$ und das Lisa-Polygon $L = (w_1, ..., w_m)$

[3] Hier Fridays-for-Future-Demonstration einfügen.

mit v_i und w_i als Eckpunkten, wobei v_1 und w_1 die Punkte mit der minimalen y-Koordinate 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_{n+1} = v_1; w_{m+1} = w_1
    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
        end if
        if WINKEL(v_i, v_{i+1}) > WINKEL(w_j, w_{j+1}) then
            j ← j + 1
        end if
        if WINKEL(v_i, v_{i+1}) = WINKEL(w_j, w_{j+1}) then
            i ← i + 1; j ← j + 1
        end if
    until i = n + 1 and j = m + 1
end function
```

Dieser Algorithmus läuft offensichtlich in $\mathcal{O}(n+m)$, denn i wird höchstens n Mal inkrementiert und j m Mal. Da er $n_{Polygon}$ Mal[4] ausgeführt wird, erhält man $\mathcal{O}(n_{Polygon} \cdot (n+m)) = \mathcal{O}(n_{Orig} + n_{Polygon} \cdot m)$, wobei n_{Orig} das ursprüngliche n als Anzahl aller Eckpunkte aller Polygone ist. Diese Erweiterung wurde **implementiert**.

1.7.4 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**.

1.7.5 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

[4] Anzahl der Polygone

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

2.2 Struktur des Programms

2.3 Implementierung der wichtigsten 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