Universität Stuttgart

Institut für Formale Methoden der Informatik

Universitätsstraße 38

70569 Stuttgart

Masterarbeit

Sichtbarkeit in triangulierten planaren Unterteilungen

Dominik Larche

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
begonnen am:	12.09.2023
beendet am:	15.12.2023

Kurzfassung

In dieser Arbeit wird ein neuer Algorithmus für eine effiziente Berechnung der sichtbaren Hindernisecken in einem euklidischen Raum mit Hindernissen vorgestellt. Dieser Algorithmus wird anschließend dazu vewendet, einerseits den vollständigen Sichtbarkeitsgraphen und andererseits mithilfe eines Dijkstra-basierten Verfahrens den kürzesten Pfad in einem euklidischen Raum mit Hindernissen zu ermitteln. Die hier vorgestellten Algorithmen werden, im Gegensatz zu den Ansätzen aus früheren Papern, auf einem klassischen Rechner implementiert und ihre Laufzeiten werden mit denen der naiven Verfahren verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

1 Einleitung

Nach Hershberger [3] existieren vielfältige praktische Anwendungen für das Finden eines kürzesten Pfades für gegebene Start- und Zielpunkte. Da dieses Problem für das Routing auf (Straßen-) Netzwerken schon seit langer Zeit bekannt ist, gibt es hierfür bereits zahlreiche Ansätze. Der Dijkstra Algorithmus bildet die Grundlage für eine effiziente Berechnung des optimalen Pfades. Zudem existieren verschiedene Ansätze zur Beschleunigung. Zu den Bekanntesten zählen u.a. die Contraction Hierarchies, das Hub Labeling sowie der A* Algorithmus. Dennoch liegt die älteste Variante des Problems in dem Finden eines kürzesten Pfades im euklidischen Raum. Bei dem Routing von Fußgängern in der Natur sowie auf großen Plätzen wird seine Relevanz verdeutlicht. Der Markusplatz in Venedig kann beispielsweise nur schwer als Netzwerk aufgefasst werden, auf welchem die Fußgänger sich nur entlang definierter Kanten fortbewegen können. Hier sind die Fußgänger durchaus dazu in der Lage, in beliebige Richtungen zu laufen, solange kein Hindernisobjekt sie blockiert. Bei der Roboterplanung in der Maschinenhalle liegt ein ähnliches Problem vor, da Roboter sich im Allgemeinen frei bewegen können und nur durch in der Halle vorkommende Hindernisse, wie etwa Arbeitsstationen oder Regale, in ihrer Bewegungsfreiheit eingeschränkt werden. Dasselbe gilt für das Routing von Schiffen. Die Gewässer des Planeten bilden kein Netzwerk mit definierten Kanten, sondern einen euklidischen Raum, welcher durch eine Vielzahl an Hindernisobjekten (Festland) begrenzt ist. Der Dijkstra Algorithmus für diskrete Graphen scheitert bei der Berechnung kürzester Pfade, da in derartigen Kontexten im Allgemeinen eine nicht zu begrenzende Anzahl von Pfaden existiert. Aus diesem Grund ist es erforderlich den vorliegenden Suchraum durch den Einsatz geeigneter Kriterien zu beschränken. Das Routing von Drohnen bzw. Raumschiffen und Satelliten im Weltall definiert ein noch deutlich komplexeres Problem, nämlich das Finden eines kürzesten Pfades im 3-dimensionalen euklidischen Raum. Canny und Reif [1] haben gezeigt, dass dieses Problem NP-Vollständig ist, daher wird im Rahmen dieser Arbeit nur der 2-dimensionale Raum betrachtet.
Als Grundannahme wird für die vorliegende Arbeit vorausgesetzt, dass eine Triangulierung der Freifläche vorliegt, wie in Abbildung 1.1 dargestellt. Somit ist der Graph $G = (V, T)$ durch seine Knotenmenge V und seine Dreiecksmenge T definiert. Jeder Knoten $v \in V$ entspricht dabei einer natürlichen Zahl, welche auf einen 2-Tupel $(v.x, v.y) \in \mathbb{R}^2$ (ihre Koordinaten im 2-dimensionalen Raum) verweist und jedes Dreieck $t \in T$ entspricht einer natürlichen Zahl, welche auf ein 3-Tupel $(v_1, v_2, v_3) \in V^3$ (3 Knoten, welche die Ecken des Dreiecks t bilden und dabei eine Reihenfolge einhalten, die gegen den Uhrzeigersinn verläuft) verweist. Sei $T(v)$ die Menge der zu v adjazenten Dreiecke in T und $T(t)$ die Menge der zu t benachbarten Dreiecke in T. Beide Mengen werden beim Einlesen des Graphen vorberechnet und können im weiteren Verlauf effizient genutzt werden.
Da keiner der SVG-Viewer eine gute Veranschaulichung ermöglichte, wurden die Bilder mit einem 3D-Viewer visualisiert.
(Link: `https://github.com/invor/simplestGraphRendering`)
Mithilfe der vorliegenden Triangulierung sollen von einem gegebenen Startpunkt aus, die sichtbaren Ecken eines jeden Hindernisses (sichtbar = es liegt kein Hindernis zwischen dem Startpunkt und der Hindernisecke) ermittelt werden.
Im Folgenden vermittelt Kapitel 2 die für den Hauptteil erforderlichen theoretischen Grundlagen

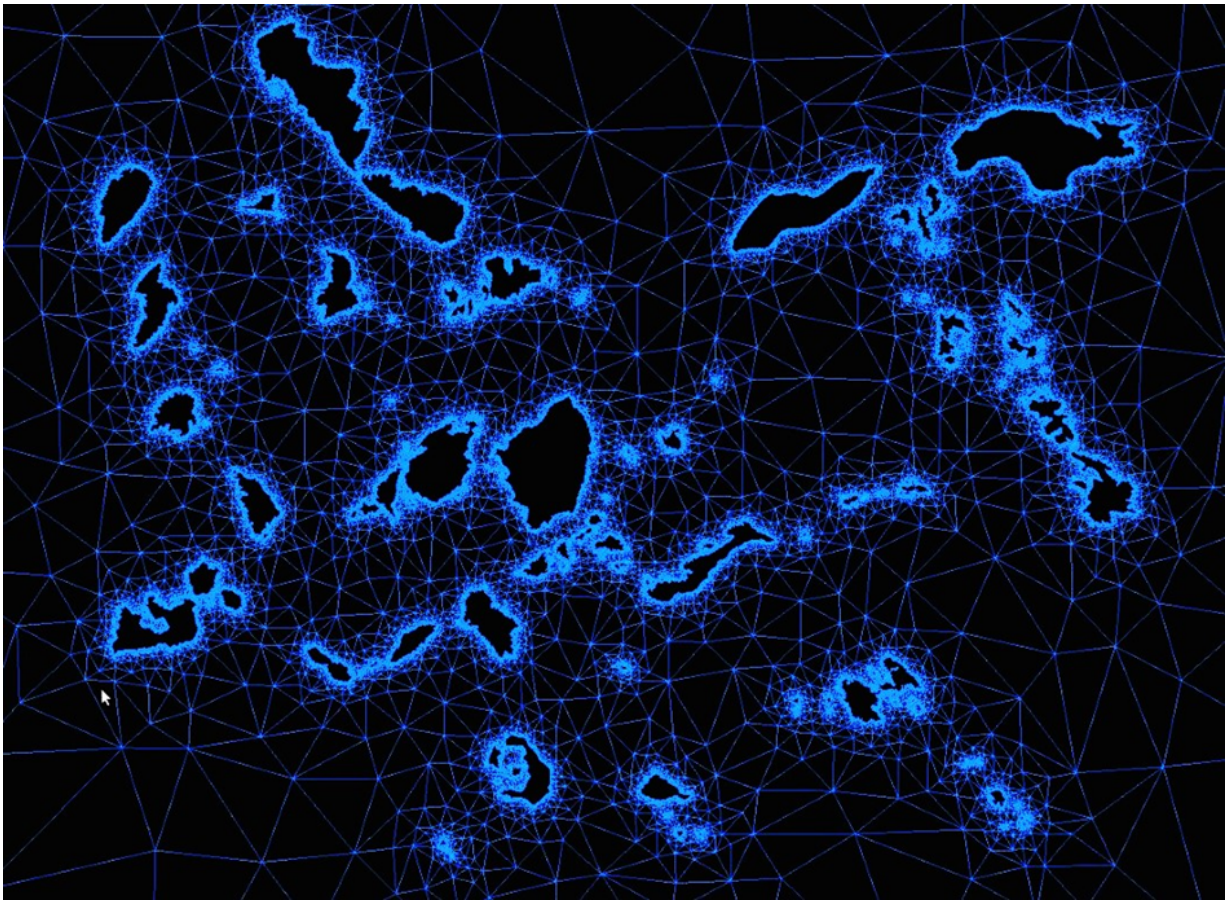


Abbildung 1.1: Visualisierung eines triangulierten Graphen mit Hindernissen, welches die Gewässer unseres Planeten (Blaue Dreiecke) und das Festland (Nicht triangulierte Hindernisse) darstellt

zu den relevanten Eigenheiten des euklidischen Raums und diverser zu diesem Kontext vorliegenden Datenstrukturen und Algorithmen. Kapitel 3 beschreibt die algorithmische Konstruktion des Sichtbarkeitsgraphen, insbesondere die Arbeitsweise des $VertexFinder$, welcher hierbei die entscheidende Rolle übernimmt. In Kapitel 4 wird ein Dijkstra-basiertes Verfahren beschrieben, welches den $VertexFinder$ für eine „On-Demand"-Konstruktion des Sichtbarkeitsgraphen nutzt. Kapitel 5 geht auf die Experimente ein, welche zu den Implementierungen der Algorithmen aus Kapitel 3 und 4 durchgeführt worden sind und erläutert ihre Ergebnisse. Kapitel 6 fasst die Arbeit sowie die darin gesammelten Erkenntnisse noch einmal zusammen und gibt einen kurzen Ausblick für das Thema.

2 Theoretische Grundlagen

In diesem Kapitel gehen wir auf die theoretischen Grundlagen zu dem euklidischen Raum, dem Sichtbarkeitsgraphen, dem Dijkstra Algorithmus, dem Kontinuierlichen Dijkstra Algorithmus sowie dem Orientierungstest ein, bevor im Hauptteil die für die Berechnungen entwickelten Algorithmen zu den jeweiligen Problemen sowie die hierzu durchgeführten Experimente mit ihren Ergebnissen erläutert werden.

2.1 Triangulierte planare Unterteilung eines euklidischen Raums mit Hindernissen

Wir betrachten kürzeste Pfade im von Hershberger [3] beschriebenen (freien) euklidischen Raum, welcher durch eine Menge von Polygonen (Hindernissen) begrenzt ist. Eines der Polygone umschließt den euklidischen Raum und begrenzt ihn auf diese Weise. Die anderen Polygone definieren die Hindernisse im Raum. Der euklidische Raum ohne Hindernispolygone ist durch eine Triangulierung planar unterteilt. Die Ecken eines jeden Polygons definieren hierbei jeweils einen Knoten. Zudem existieren weitere Knoten im euklidischen Raum. Die Triangulierung setzt sich aus einer Menge von Dreiecken zusammen, wobei ein Dreieck durch jeweils drei Knoten definiert ist. Die Dreiecke sind so gewählt, dass eine planare Unterteilung vorliegt, zwei verschiedene Dreieckskanten sich demnach nicht schneiden. In Abbildung 1.1 ist dies veranschaulicht.
Seien nun zwei Punkte s und t mit ihren jeweiligen Koordinaten $(s.x, s.y, t.x, t.y)$ im begrenzten euklidischen Raum gegeben. Das Ziel besteht darin den kürzesten Pfad π von s nach t, welches keines der Hindernispolygone durchquert, so effizient wie möglich zu berechnen. Dabei ist zu beachten, dass ein beliebiger Pfad im euklidischen Raum entweder direkt von s nach t führt oder an mindestens einer Stelle seine Richtung ändert. Liegt eine dieser Stellen nicht an der Hindernisecke eines Polygons, so existiert ein alternativer Pfad, der höchstens dieselbe Länge hat und ausschließlich an Hindernisecken seine Richtung ändert. Mit anderen Worten existiert zu zwei beliebigen voneinander erreichbaren Punkten s und t mindestens ein kürzester Pfad, der nur an Ecken eines Hindernispolygons seine Richtung ändert. Dieser Pfad wird im Folgenden als $\pi(s, t)$ bezeichnet.

2.2 Sichtbarkeitsgraph

Nach Hershberger [3] benutzen frühere Ansätze den Sichtbarkeitsgraphen, um den kontinuierlichen euklidischen Raum durch einen diskreten Graphen zu ersetzen. Die Knotenmenge des Sichtbarkeitsgraphen besteht aus sämtlichen Ecken eines jeden Hindernispolygons. Zwei Knoten sind im Sichtbarkeitsgraphen durch jeweils eine Kante miteinander verbunden, wenn die beiden zugehörigen Hindernisecken sichtbar voneinander sind. Zwei beliebige Punkte a und b im euklidischen Raum sind sichtbar voneinander, wenn der direkte Pfad von a nach b (also der Pfad, welcher zu keinem Zeitpunkt seine Richtung ändert) keines der Hindernispolygone durchquert. Das Gewicht einer solchen Kante entspricht der euklidischen Distanz der beiden Hindernisecken

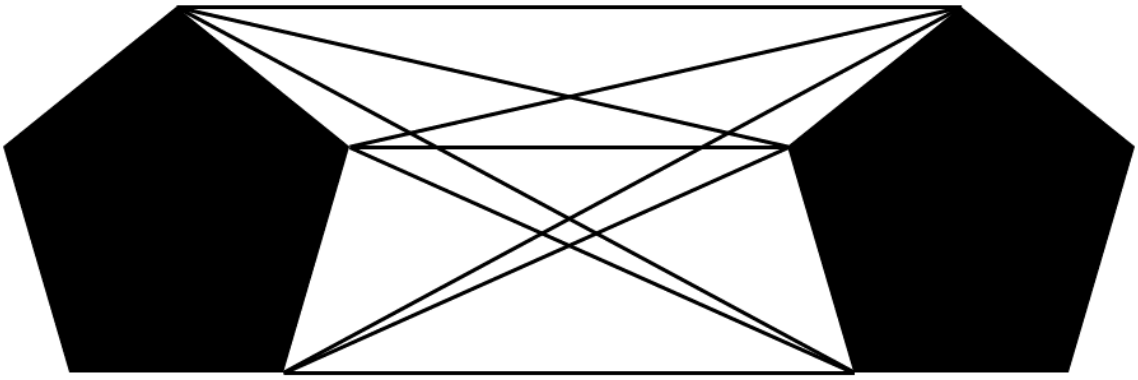


Abbildung 2.1: Sichtbarkeitsgraph für zwei Beispielpolyone

zueinander. In Abbildung 2.1 sind die gezogenen Kanten eines Sichtbarkeitsgraphen für zwei Beispielpolygone veranschaulicht.

Die Punkte s und t werden zur Berechnung des kürzesten Pfades als Knoten in den Sichtbarkeitsgraphen eingefügt. Diese beiden Knoten werden anschließend mit sämtlichen von ihnen aus sichtbaren Hindernisecken durch jeweils eine Kante miteinander verbunden. Ein kürzester Pfad $\pi(s,t)$ im euklidischen Raum, wie er im vorigen Abschnitt definiert wurde, entspricht nun einem kürzesten Pfad im Sichtbarkeitsgraphen, da dieser nur an Ecken eines Hindernispolygons seine Richtung ändert.

Ein mögliches Verfahren, um den kürzesten Pfad zweier Punke s und t im euklidischen Raum zu bestimmen, besteht nun darin den Sichtbarkeitsgraphen auszurechnen, um anschließend den Dijkstra Algorithmus zur Berechnung kürzester Pfade in einem diskreten Graphen anzuwenden. Sei n die Anzahl der Knoten und m die Anzahl der Kanten im Sichtbarkeitsgraphen. Durch den Algorithmus von Gosh und Mount [2] kann der Sichtbarkeitsgraph mit einer Laufzeit von $O(nlog(n) + m)$ bestimmt werden. Der Dijkstra Algorithmus besitzt ebenfalls eine Laufzeit von $O(nlog(n) + m)$. Folglich ist dies auch die Laufzeit des beschriebenen Algorithmus zur Berechnung kürzester Pfade im euklidischen Raum. Dieser hat im worst-case quadratische Laufzeit, da die Anzahl der Kanten im Sichtbarkeitsgraphen in der Größenordnung von $\theta(n^2)$ liegen kann. Ein Sichtbarkeitsgraph für sämtliche Gewässer des Planeten, wobei das Festland die Menge der Hindernispolygone definiert, kann aufgrund ihrer Größe nicht abgespeichert und folglich auch nicht zielgerichtet eingesetzt werden.

2.3 Dijkstra Algorithmus

In einem gewichteten Graphen $G = (V, E, c)$ (wie dem Sichtbarkeitsgraphen aus dem vorigen Abschnitt) soll der kürzeste Pfad von einem Startknoten $s \in V$ zu einem Zielknoten $t \in V$ bestimmt werden. Der folgende Pseudo-Code beschreibt den hierzu vorliegenden Algorithmus von Dijkstra:

Algorithm 1 DIJKSTRA(G, s, t)

```
for v ∈ V do
  distances[v] ← ∞
end for
distances[s] ← 0
priorityQueue.insertOrUpdate(s, 0)
while NOT priorityQueue.isEmpty() do
  v ← priorityQueue.popMin()
  for e = (v, w) ∈ E do
    if distances[v] + c(e) < distances[w] then
      distances[w] ← distances[v] + c(e)
      priorityQueue.insertOrUpdate(w, distances[w])
      previousShortestPathVertex[w] ← v
    end if
  end for
end while
return previousShortestPathVertex =0
```

Da im zurückgegebenen Array für jeden Knoten $w \in V$ der vorherige adjazente Knoten $v \in V$ auf dem kürzesten Pfad von s nach w eingetragen ist, kann der kürzeste Pfad von s nach t sowie seine Distanz einfach berechnet werden, indem von t aus beginnend, sämtliche auf dem kürzesten Pfad liegende Knoten in entgegengesetzter Reihenfolge ermittelt werden, bis der Knoten s erreicht ist.
Die Priority Queue in dem oben beschriebenen Algorithmus gibt beim Aufruf der popMin-Methode immer denjenigen Knoten zurück, welcher unter allen in der Priority Queue befindlichen Knoten die kürzeste Distanz zum Startknoten besitzt. Im A* Algorithmus, einer Variante des Dijkstra Algorithmus, wird der oben beschriebene Algorithmus durch ein weiteres Array ergänzt. Dieses Array enthält zu jedem Knoten $v \in V$ eine modifizierte Distanz. Beim Aufruf der popMin-Methode gibt die Priority Queue nun denjenigen Knoten zurück, welcher unter allen in der Priority Queue befindlichen Knoten die kürzeste modifizierte Distanz besitzt. Im Standardfall beträgt die modifizierte Distanz eines jeden Knotens $v \in V$ die Summe aus der zu dem Zeitpunkt ermittelten kürzesten Distanz von s nach v und der euklidischen Distanz von v nach t. Die Optimalität der Ergebnisse bleibt dabei unberührt. Diese Heuristik kann helfen den herkömmlichen Dijkstra Algorithmus zu beschleunigen, indem Knoten mit einer „höheren Relevanz“ früher bearbeitet werden.

2.4 Kontinuierlicher Dijkstra Algorithmus

Da die Konstruktion sowie die Anwendung eines Sichtbarkeitsgraphen zur Berechnung kürzester Pfade im worst-case eine quadratische Laufzeit benötigt, schlug Mitchell [5] einen alternativen Ansatz vor. Die Intuition dahinter besteht darin, dass eine Wellenfront sich vom Startpunkt s aus in alle Richtungen ausbreitet. Die Wellenfront ist zu jedem beliebigen Zeitpunkt δ durch die Menge aller Punkte p definiert, für welche $dist(s, p) = \delta$ gilt. Sobald die Wellenfront auf eine Hindernisecke trifft, breitet sich eine kleinere Wellenfront von diesem Knoten aus. Jeder kürzeste Pfad ist eine Sequenz von kleineren Wellenfronten, die alle jeweils in einer der Hindernisecken ihren Ursprung haben. In Abbildung 2.2 ist die Intuition hinter dem Verfahren veranschaulicht.
Sei n die Anzahl an Hindernisecken. Mitchell [5] benutzt den kontinuierlichen Dijkstra Algorithmus, um kürzeste Pfade unter der L1-Metrik (Manhattan-Metrik) mit einer Komplexität von $O(nlog(n))$ zu berechnen. Später erweitert Mitchell [6] seinen Ansatz, um kürzeste Pfade unter der L2-Metrik (Euklidische Norm) mit einer Komplexität von $O(n^{\frac{5}{3}+\epsilon})$ zu berechnen, wobei ϵ beliebig klein gewählt werden kann. Hershberger und Suri [4] präsentieren eine alternative Nutzung des kontinuierlichen Dijksta Algorithmus, welche den kürzesten Pfad im euklidischen Raum mit einer Komplexität von $O(nlog(n))$ berechnet. Die genannten Verfahren sind algorithmisch korrekt und effizient, jedoch sehr kompliziert und programmiertechnisch kaum realisierbar.

2.5 Orientierungstest im 2-dimensionalen Raum

Seien drei Punkte a, b und p im 2-dimensionalen Raum gegeben. Der Orientierungstest im 2-dimensionalen Raum definiert eine Funktion, welche die drei genannten Punkte mit ihren jeweiligen Koordinaten $(a.x, a.y, b.x, b.y, p.x, p.y)$ als Eingabe nimmt und einen Zahlenwert $n \in \{-1, 0, 1\}$ zurückliefert, der angibt auf welcher Seite der durch die beiden ersten Punkte a und b definierten Geraden, der dritte Punkt p sich befindet.
$n = -1 \implies$ Der Punkt p befindet sich links von der Geraden von a nach b.
$n = 1 \implies$ Der Punkt p befindet sich rechts von der Geraden von a nach b.
$n = 0 \implies$ Der Punkt p befindet sich auf der Geraden von a nach b.

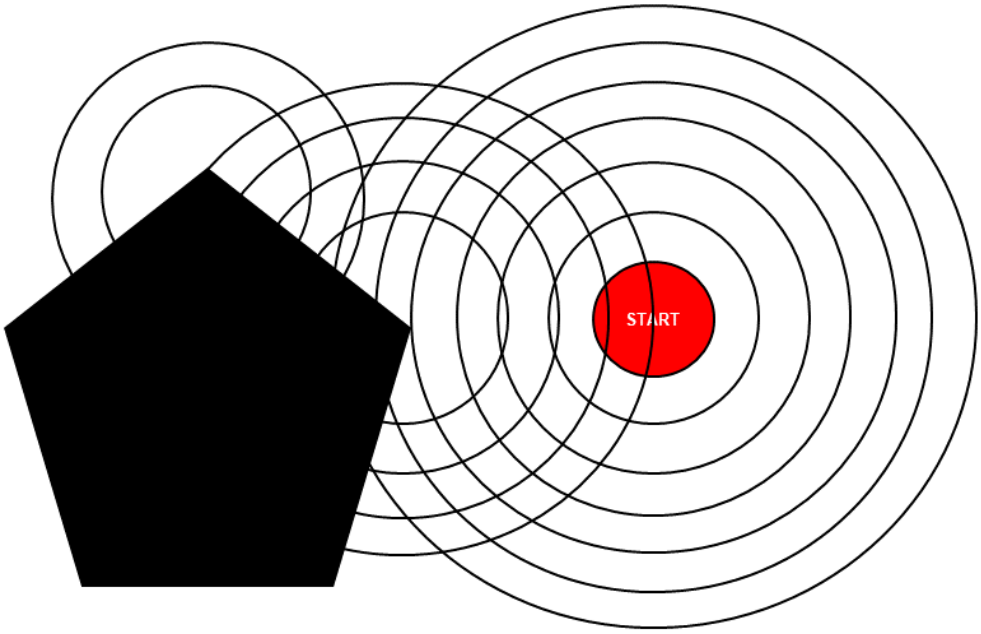


Abbildung 2.2: Veranschaulichung der Intuition hinter dem kontinuierlichen Dijkstra Verfahren

In Abbildung 2.3 ist ein Beispiel für den Orientierungstest im 2-dimensionalen Raum veranschaulicht.
Der folgende Pseudo-Code beschreibt die hierzu durchgeführte Berechnung.
Sämtliche angewendeten geometrischen Operationen sind jeweils als eine Abfolge von low-level Operationen und Orientierungstests im 2-dimensionalen Raum implementiert worden.

Algorithm 2 ORIENTATIONTEST(a, b, p)

$b.x \leftarrow b.x - a.x$
$b.y \leftarrow b.y - a.y$
$p.x \leftarrow p.x - a.x$
$p.y \leftarrow p.y - a.y$
$crossProduct \leftarrow b.x \cdot p.y - b.y \cdot p.x$
if $crossProduct > 0$ **then**
  return -1
else if $crossProduct < 0$ **then**
  return 1
else
  return 0
end if=0

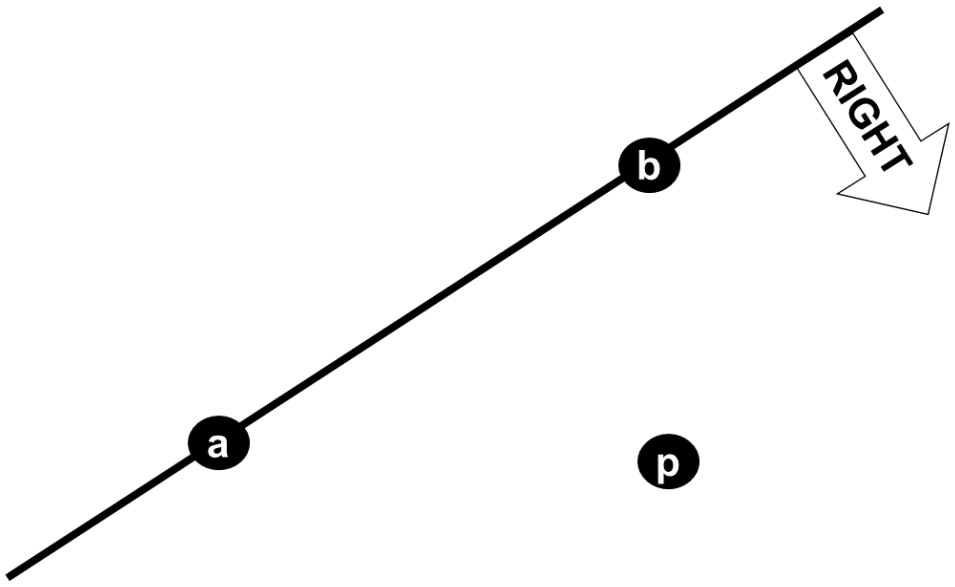


Abbildung 2.3: Beispiel für den Orientierungstest im 2-dimensionalen Raum

3 Sichtbarkeitsgraph

In diesem Kapitel wird zunächst die naive Konstruktion des Sichtbarkeitsgraphen und anschließend ihre Konstruktion unter Verwendung des $VertexFinder$ erläutert.

3.1 Naive Konstruktion

Im ersten Schritt iteriert der Algorithmus über sämtliche Knoten im Graphen $G = (V, T)$ und prüft für jeden einzelnen $v \in V$, ob es sich um eine Hindernisecke handelt.
Um dies zu ermitteln, wird die Routine $isObstacle$ aufgerufen, welche für einen gegebenen Knoten $v \in V$ untersucht, ob sie Ecke eines Hindernispolygons ist. Die Routine iteriert über sämtliche Dreiecke $t \in T(v)$ und bestimmt die Anzahl $|T(t)|$ der Nachbardreiecke für t. Im Fall $|T(t)| < 2$ ist der Knoten v definitiv die Ecke eines Hindernispolygons. Für $|T(t)| = 2$ wird eine Fallunterscheidung benötigt. Hierzu wird im Anschluss ermittelt, ob der Knoten v für die beiden Nachbardreiecke jeweils einen Eckpunkt repräsentiert. Ist dies nicht der Fall, so ist v ebenfalls die Ecke eines Hindernispolygons. Andernfalls und wenn $|T(t)| = 3$ gilt, wird keine Aussage getroffen und das nächste Dreieck in $T(v)$ betrachtet. Liegt nach Behandlung sämtlicher Dreiecke in $T(v)$ kein Beweis dafür vor, dass v eine Hindernisecke repräsentiert, so handelt es sich bei v um keine Hindernisecke.
Die Hindernisecken werden in ein Array geschrieben und bilden die Knotenmenge für den Sichtbarkeitsgraphen.
Nun folgt eine doppelte Schleife über alle Hindernisecken, in der für jeweils zwei Hindernisecken $v, w \in V$ ermittelt wird, ob diese beiden Knoten sichtbar voneinander sind. Ist dies der Fall, so wird im zu konstruierenden Sichtbarkeitsgraphen eine Kante von v nach w und eine Kante von w nach v hinterlegt. In Abbildung 3.1 wird das Sichtbarkeitskriterium für jeweils zwei Hindernisecken veranschaulicht.
Um die Sichtbarkeit zu prüfen, wird die Routine $isVisible$ aufgerufen, welche für zwei gegebene Knoten $v, w \in V$ untersucht, ob diese voneinander sichtbar sind. Hierzu startet die Routine bei einem Dreieck $t \in T(v)$, welches zu Beginn das aktuell betrachtete Dreieck bezeichnet. Nun wird dasjenige Nachbardreieck aus $T(t)$ ermittelt, dessen gemeinsame Kante mit t das durch die beiden Knoten v und w definierte Segment s schneidet. Dieses Nachbardreieck repräsentiert nun das aktuell betrachtete Dreieck. Von ihm aus wird die Prozedur weiter fortgesetzt, bis ein Dreieck in $T(w)$ erreicht wird. In diesem Fall sind die beiden Knoten v und w voneinander sichtbar. Kann im Laufe der Prozedur mal kein noch nicht bearbeitetes Dreieck in $T(t)$ ausfindig gemacht werden, dessen gemeinsame Kante mit dem aktuell betrachteten Dreieck das Segment s schneidet, sind die beiden Knoten v und w nicht voneinander sichtbar.
Eine algorithmisch naivere Methode die Sichtbarkeit zwischen zwei Hindernisecken zu ermitteln, besteht darin für sämtliche Hinderniskanten zu überprüfen, ob mindestens eine unter ihnen von dem Segment s geschnitten wird. Dies ist jedoch sehr aufwendig zum Implementieren, da hierzu einige Sonderfälle berücksichtigt werden müssen. Darüber hinaus ist die Laufzeit deutlich schlechter im Vergleich zu der von der Routine $isVisible$. Aus diesem Grund wird die algorithmisch naivere Methode nicht weiterverfolgt.

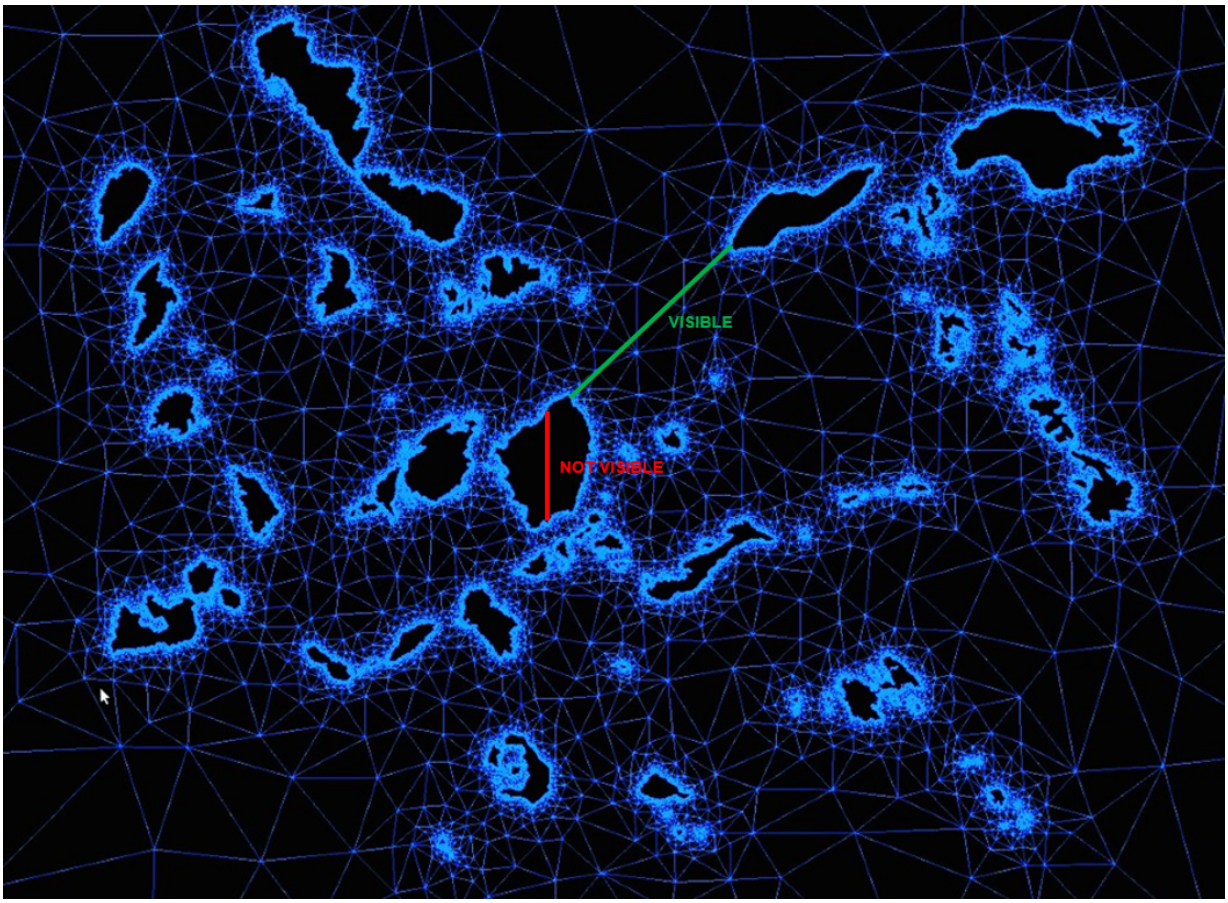


Abbildung 3.1: Sichtbarkeitskriterium je zweier Hindernisecken für die naive Konstruktion eines Sichtbarkeitsgraphen

3.2 Vertex Finder

Der $VertexFinder$ berechnet für einen gegebenen Startknoten $v_{start} \in V$ oder für einen gegebenen Startpunkt p_{start} mit seinen Koordinaten $(p_{start}.x, p_{start}.y)$ bis zu einer vorgegebenen Distanz alle Hindernisecken die von v_{start} bzw. p_{start} aus sichtbar sind. Hierbei wird, im Gegensatz zum naiven Verfahren, jedes Dreieck nur einmal untersucht, sodass die Gesamtzahl an Bearbeitungen für n Dreiecke in $O(n)$ liegt.
Der Algorithmus beginnt bei einem Dreieck $t_{start} \in T$, das v_{start} bzw. p_{start} enthält. Während seiner Bearbeitung werden die Dreiecke $t \in T$, mit denen t_{start} mindestens einen Knoten gemeinsam hat, in eine Datenstruktur mit dem Namen *triangleQueue* eingefügt. Es wird ein Dreieck nach dem anderen aus *triangleQueue* bearbeitet und während der Bearbeitung eines Dreiecks werden ihre umliegenden noch nicht bearbeiteten Dreiecke in *triangleQueue* aufgenommen. Bei der Bearbeitung eines Dreiecks $t \in T$ werden die Sichteinschränkungen von der / den v_{start} bzw. p_{start} zugeneigten Kante(n) auf die übrige(n) Kante(n) übertragen. Eine Kante ist v_{start} bzw. p_{start} zugeneigt, wenn v_{start} bzw. p_{start} sich rechts von ihr befindet. (Wie in Kapitel 1 beschrieben, werden die 3 Knoten eines Dreiecks in einer Reihenfolge gespeichert, die gegen den Uhrzeigersinn verläuft.) Dies wird mit einem einfachen Orientierungstest im 2-dimensionalen Raum ermittelt. Eine Sichteinschränkung definiert das von v_{start} bzw. p_{start} aus durch ein auf dem direkten Weg befindliches Hindernis verdeckte Intervall einer spezifischen Dreieckskante. Aus diesem Grund wird eine Sichteinschränkung je Dreieck je Kante in einer Datenstruktur mit dem Namen *constraintList* notiert. Eine einzelne Sichteinschränkung c wird dabei durch zwei Knoten $c = (a, b)$ definiert. Der Schnittpunkt der durch v_{start} bzw. p_{start} und a definierten Geraden mit der Kante e, für die jene Sichteinschränkung c notiert wurde, definiert den Beginn der Sichteinschränkung c auf der Kante e. Der Schnittpunkt der durch v_{start} bzw. p_{start} und b definierten Geraden mit der Kante e, für die jene Sichteinschränkung c notiert wurde, definiert das Ende der Sichteinschränkung c auf der Kante e. In Abbildung 3.2 wird bei der Bearbeitung des Dreiecks t_6 die Sichteinschränkung (rotes Segment) von der an t_4 grenzenden Kante auf die an t_7 grenzende Kante übertragen. t_1, t_4 sowie t_2 sind bereits bearbeitet worden. t_3 und t_7 haben die höchste Priorität, können also gleich nach der Bearbeitung von t_6 selbst bearbeitet werden. t_5 darf nicht bearbeitet werden, solange t_3 noch nicht bearbeitet wurde, da die Sichteinschränkungen auf der gemeinsamen Kante von t_3 und t_5 erst durch die Bearbeitung des Dreiecks t_3 ermittelt werden müssen, bevor diese auf die anderen beiden Kanten von t_5 übertragen werden können. t_3 ist somit relevant für t_5. (Die Relevanz eines Dreiecks für ein benachbartes Dreieck wird im weiteren Verlauf von Bedeutung sein.) Im Folgenden werden die Einzelheiten des $VertexFinder$ näher erläutert.

3.2.1 Vorbereitung

Zuallererst wird das Startdreieck bzw. werden die Startdreiecke ermittelt. Hierzu nimmt der Algorithmus entweder alle Dreiecke in $T(v_{start})$ oder berechnet für p_{start} dasjenige Dreieck $t_{start} \in T$, das den Punkt p_{start} enthält. Im Letzteren der beiden Fälle wird die Routine $findTriangle$ aufgerufen, welche über sämtliche Dreiecke $t \in T$ iteriert und für jedes dieser Dreiecke prüft, ob der Punkt p_{start} in t enthalten ist. Sobald ein Dreieck t den Punkt p_{start} enthält, wird dieses zurückgegeben. Sollte keines der Dreiecke in T den Punkt p_{start} enthalten, so wird der Wert -1 zurückgegeben. Um zu überprüfen, ob ein bestimmtes Dreieck t den Punkt p_{start} enthält, wird die Routine *pointInTriangle* aufgerufen, die für einen gegebenen Punkt p und ein gegebenes Dreieck t ermittelt, ob p in t enthalten ist. Hierzu wird für jede der drei Kanten in t die unterstützende Gerade betrachtet. Der Orientierungstest im 2-dimensionalen Raum untersucht anschließend, ob

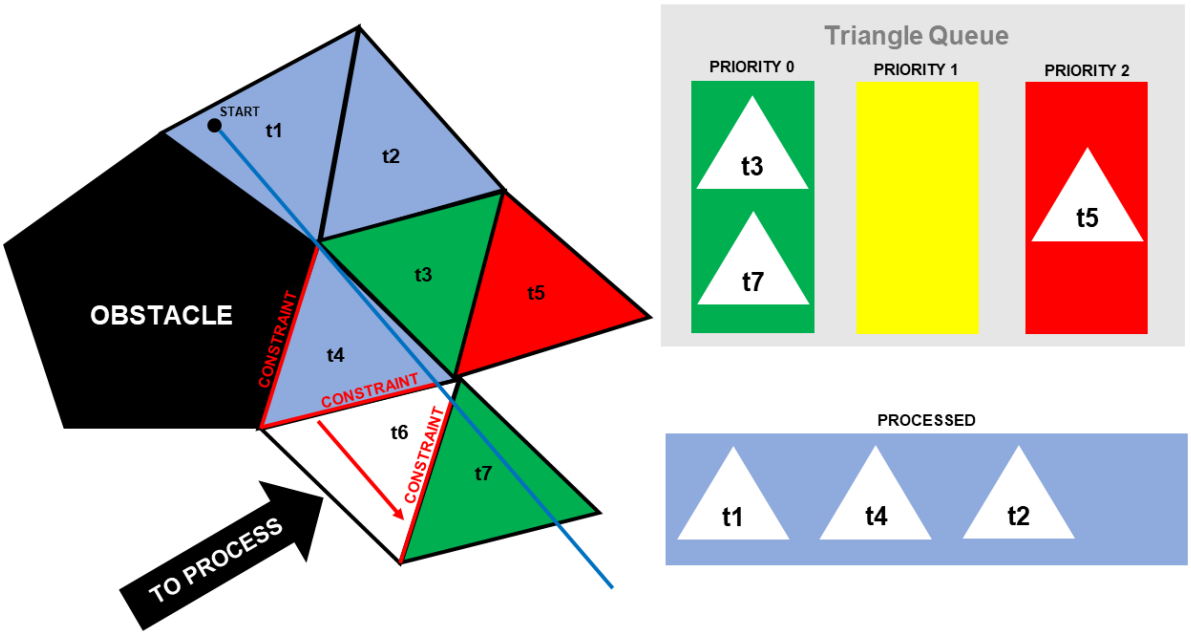


Abbildung 3.2: Allgemeiner Ablauf des $VertexFinder$

der Punkt p sich jeweils auf bzw. links von der Geraden befindet. Ist dies für jede der drei Kanten der Fall, so liegt p in t, andernfalls nicht.
Nun werden die für den $VertexFinder$ erforderlichen Datenstrukturen aufgesetzt.
$relevantTrianglesEdges$ hat die Aufgabe für ein gegebenes Dreieck die für ihn zur Bearbeitung relevanten Dreiecke sowie die gemeinsame Kante der beiden Dreiecke zu notieren. Für ein Dreieck $t \in T$ sind die relevanten Dreiecke jene, die für eine wirkungsvolle Bearbeitung von t zuvor bearbeitet sein müssen. Mit anderen Worten: Solange seine relevanten Dreiecke noch nicht bearbeitet wurden, sollte t nicht bearbeitet werden. Dies gilt jedoch nicht immer, was im weiteren Verlauf des Algorithmus noch näher erläutert wird. In Abbildung 3.3 ist ein Beispiel für die Beziehung zwischen einem Dreieck und seinem relevanten Dreieck veranschaulicht.
$triangleQueue$ sammelt während der Ausführung des Algorithmus die zu bearbeitenden Dreiecke ein und ordnet sie nach ihrer Priorität. Höher priorisierte Dreiecke werden zur Bearbeitung vorgezogen. Die Priorisierung in $triangleQueue$ nutzt die Informationen aus $relevantTrianglesEdges$. Ein Dreieck, das auf keine relevanten Dreiecke wartet oder dessen relevanten Dreiecke alle bereits bearbeitet wurden, erhält die Priorität 0 (die höchste Priorität). Ein Dreieck, das auf die Bearbeitung mindestens eines relevanten Dreieckes $t \in T$ wartet, wobei t sich bereits in $triangleQueue$ befindet und selbst auf die eigene Bearbeitung wartet, erhält die Priorität 2 (die niedrigste Priorität). Ein Dreieck, das auf die Bearbeitung mindestens eines relevanten Dreieckes wartet, wobei sich keines seiner relevanten Dreiecke in $triangleQueue$ befindet, erhält die Priorität 1 (die mittlere Priorität). Nach dem Einfügen eines Dreiecks muss die Priorität während seiner Aufenthaltsphase in $triangleQueue$ u.U. mehrmals aktualisiert werden. In Abbildung 3.4 ist das Konzept hinter der Priorisierung innerhalb von $triangleQueue$ beispielhaft veranschaulicht.
Das Dreieck 6 wurde bereits bearbeitet, das Dreieck 9 ist noch zu keinem Zeitpunkt in $triangleQueue$ enthalten gewesen. Das Dreieck 3 hat die höchste Priorität, da ihr relevantes Dreieck 6 bereits bearbeitet wurde und 3 folglich auf kein relevantes Dreieck mehr warten muss. Das Dreieck 7

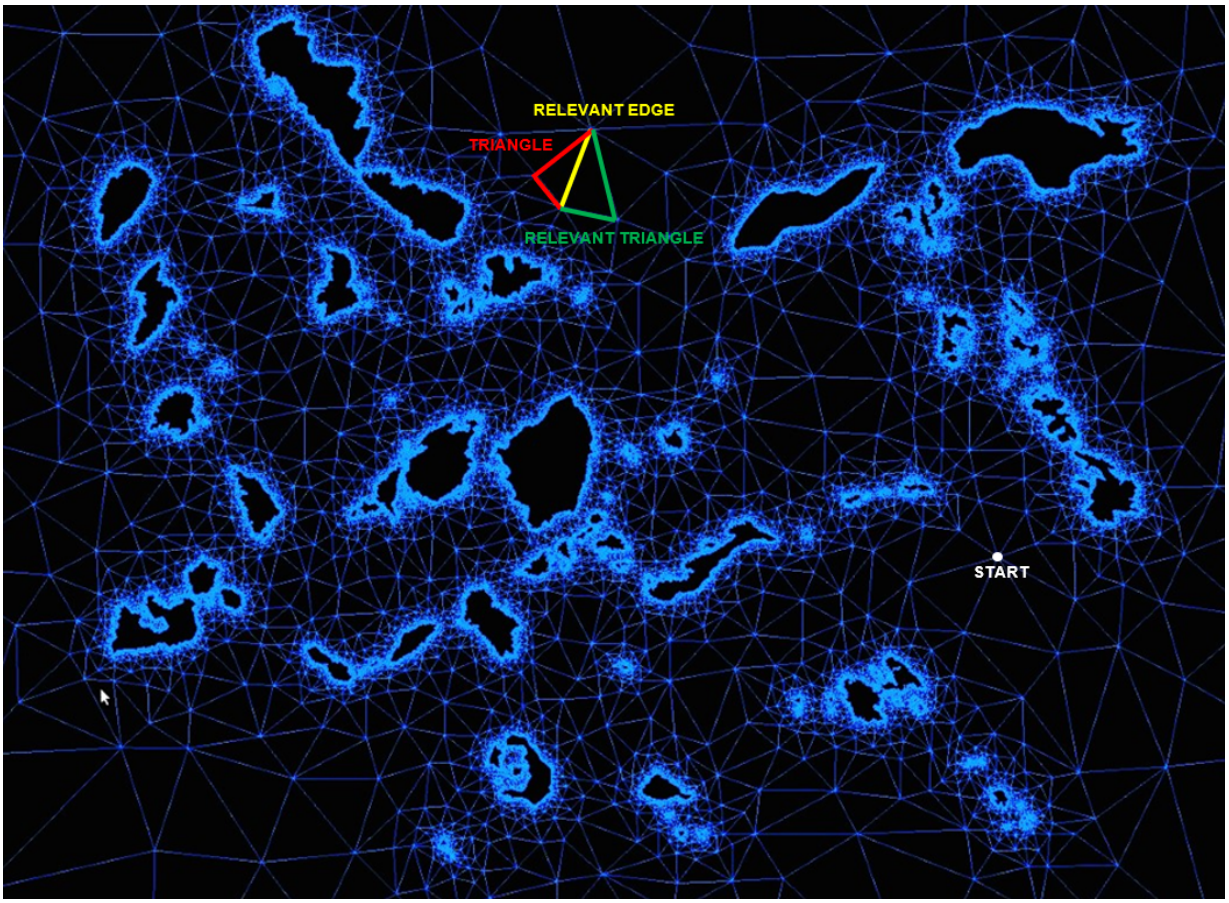


Abbildung 3.3: Ein Dreieck mit seiner dem Startknoten zugeneigten Kante und seinem relevanten Dreieck

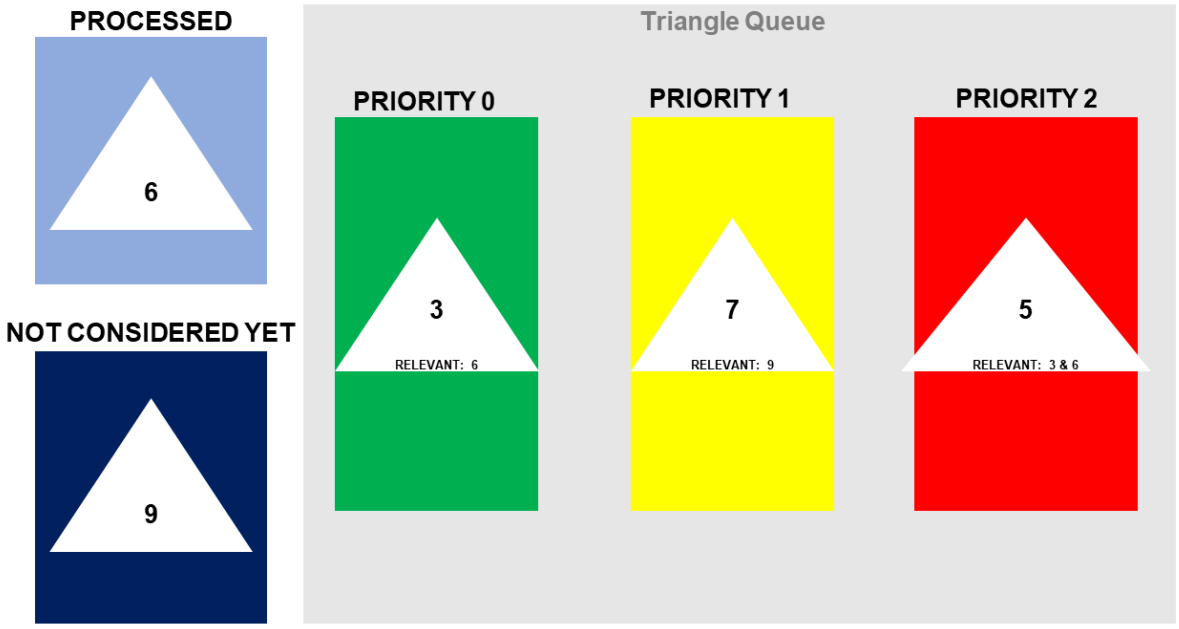
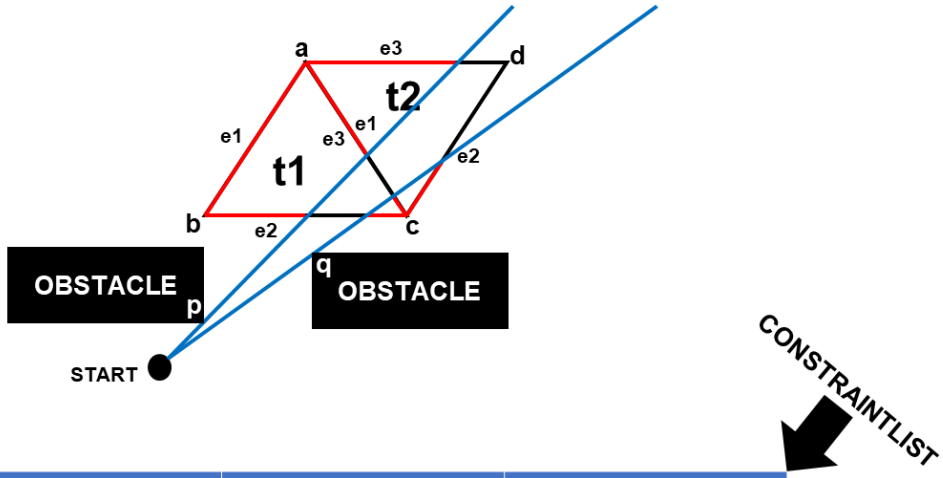


Abbildung 3.4: Priorisierung der Dreiecke innerhalb von *triangleQueue*

hat mittlere Priorität, da ihr relevantes Dreieck 9 zwar noch nicht bearbeitet worden ist, es sich jedoch auch nicht in *triangleQueue* befindet. Das Dreieck 5 erhält die niedrigste Priorität, da ihr relevantes Dreieck 3 sich in *triangleQueue* befindet und auf die eigene Bearbeitung wartet. Dass das bereits bearbeitete Dreieck 6 ebenfalls ein für 5 relevantes Dreieck darstellt, spielt hierbei keine Rolle.
constraintList notiert während der Ausführung des Algorithmus sämtliche Sichteinschränkungen einer jeden Dreieckskante bis zu der dem $VertexFinder$ vorgegebenen Distanz zu v_{start} bzw. p_{start}. Jede Sichteinschränkung wird dabei durch zwei Knoten $a, b \in V$ charakterisiert. a repräsentiert den Beginn, b das Ende der Sichteinschränkung. Die Sichteinschränkungen werden je Dreieck, je Kante festgehalten. Da eine Kante zu zwei unterschiedlichen Dreiecken gehören kann, pflegt *constraintList* eine Look-Up Tabelle durch welche die Informationen zu den Sichteinschränkungen der Kante eines Dreiecks effizient anhand der Informationen zu den Sichteinschränkungen derselben Kante eines anderen Dreiecks abgelesen werden können. Eine eingefügte Sichteinschränkung wird in die vorhandene Sortierung der Sichteinschränkungen der betroffenen Dreieckskante gemäß der Lage ihrer repräsentierenden Knoten eingeordnet, damit überlappende Sichteinschränkungen verschmolzen werden können. Das ist notwendig, um die Anzahl der zu speichernden Sichteinschränkungen gering zu halten. In Abbildung 3.5 ist die Arbeitsweise von *constraintList* beispielhaft veranschaulicht.

Für das Dreieck t_1 definieren die beiden Knoten b und a die einzige Sichteinschränkung der Kante e_1. Das Segment definiert durch den Startpunkt und b verläuft links von dem Segment definiert durch den Startpunkt und a, daher wird b in der Sortierung vor a eingeordnet. Die beiden Knoten b und p definieren die erste Sichteinschränkung, die beiden Knoten q und c definieren die zweite Sichteinschränkung auf der Kante e_2. Die Sortierung der vier Knoten erfolgt anhand desselben Kriteriums wie zuvor für die beiden Knoten a und b auf der Kante e_1. Wie bereits erwähnt, werden überlappende Sichteinschränkungen miteinander verschmolzen. Für die Kante



EDGE \ TRIANGLE	t1	t2
e1	b,a	LOOKUP[t1][e3]
e2	b,p,q,c	q,c
e3	a,p,q,c	a,p

Abbildung 3.5: Beispielhafte Hinterlegungen von Sichteinschränkungen in *constraintList*

e_1 des Dreiecks t_2 ist ein Look-Up hinterlegt, welcher auf die Kante e_3 des Dreiecks t_1 verweist. Es handelt sich hierbei nämlich um dieselbe Kante und da t_1 als ein für t_2 relevantes Dreieck zu verzeichnen ist, wurden die Sichteinschränkungen dieser Kante bereits bei der Bearbeitung von t_1 hinterlegt, bevor die Bearbeitung von t_2 begonnen hat.
constraintList ermöglicht zudem das Aufsammeln der sichtbaren Hindernisecken sämtlicher im Verlauf des Algorithmus bearbeiteten Dreiecke. Im letzten Schritt der Vorbereitung wird das Startdreieck bzw. eines der Startdreiecke in *triangleQueue* eingefügt. Dieses erhält natürlich die Priorität 0. Nun beginnt eine Schleife über jedes aus *triangleQueue* gezogene Dreieck, wobei *triangleQueue* im Laufe einer Iteration mit weiteren Dreiecken versorgt wird.

3.2.2 Erste Schritte

Zuallererst wird ein Dreieck $t_{current} \in T$ mit der Priorität 0 aus *triangleQueue* gezogen. Existiert kein Dreieck mit der Priorität 0, so wird ein Dreieck mit der Priorität 1 gezogen. In diesem Fall sucht *triangleQueue* nach demjenigen Dreieck $t_{current} \in T$ mit der Priorität 1, das die kürzeste Distanz zu v_{start} bzw. p_{start} besitzt. Dieses Dreieck wird gezogen und es wird notiert, dass jenes Dreieck noch auf mindestens ein relevantes Dreieck wartet, welches noch zu keinem Zeitpunkt in *triangleQueue* enthalten war. Wie weiter oben erwähnt, ist dieser Fall möglich und wird im weiteren Verlauf von Bedeutung sein. Existiert kein Dreieck mit der Priorität 0 und auch kein Dreieck mit der Priorität 1, ist *triangleQueue* leer, da es nicht möglich ist, dass alle Dreiecke in *triangleQueue* die Priorität 2 besitzen, sofern mindestens eines enthalten ist. Das liegt an der Konvexität der Dreiecke und die daraus resultierende azyklische Beziehung der Relevanz unter den Dreiecken.
Nun wird geprüft, ob $t_{current}$ sich innerhalb der dem $VertexFinder$ vorgegebenen Distanz zu v_{start} bzw. p_{start} befindet. Liegt $t_{current}$ zu weit entfernt, so wird die Bearbeitung beendet und

das nächste Dreieck aus *triangleQueue* gezogen. (Eine vorgegebene Distanz von 0.0 wird so behandelt, als würde gar keine Einschränkung in der Distanz vorliegen und folglich wird die Bearbeitung von $t_{current}$ nicht beendet.)
Wurde seine Bearbeitung nicht beendet, so wird $t_{current}$ in *constraintList* berücksichtigt. Dies ist für das Aufsammeln der sichtbaren Hindernisecken am Ende von Bedeutung.

3.2.3 Aufsammeln der adjazenten Dreiecke

Damit der Algorithmus nicht nach der ersten Iteration seine Arbeit beendet, ist es erforderlich *triangleQueue* mit weiteren Dreiecken zu versorgen. Hierzu wird für alle drei Knoten v des Dreiecks $t_{current}$ jedes Dreieck $t \in T(v)$ betrachtet. Ist t bereits in *triangleQueue* enthalten bzw. zu einem früheren Zeitpunkt in *triangleQueue* enthalten gewesen, so wird es übersprungen. Andernfalls werden die für t relevanten Dreiecke in $relevantTrianglesEdges$ hinterlegt. Hierzu wird für jedes Nachbardreieck $t_{candidate} \in T(t)$ ihre mit t gemeinsame Kante ermittelt und untersucht, ob die Kante dem Startknoten v_{start} bzw. dem Startpunkt p_{start} zugeneigt ist. Eine Kante im Dreieck t ist v_{start} bzw. p_{start} zugeneigt, wenn sich v_{start} bzw. p_{start} rechts von der unterstützenden Geraden der jeweiligen Kante befindet. Dies wird mit einem einfachen Orientierungstest im 2-dimensionalen Raum geprüft. Wenn die besagte Kante v_{start} bzw. p_{start} zugeneigt ist, so wird $t_{candidate}$ als ein für t relevantes Dreieck in $relevantTrianglesEdges$ hinterlegt, zusammen mit einem Index aus $\{0, 1, 2\}$, welcher jene Kante in $t_{candidate}$ repräsentiert. Anschließend wird t in *triangleQueue* mit aufgenommen. U.u. erhält t nicht die höchste Priorität.

3.2.4 Bearbeiten eines Startdreiecks

Wenn es sich bei $t_{current}$ um das Startdreieck bzw. um eines der Startdreiecke handelt, so existieren für die Kanten von $t_{current}$ keinerlei Sichteinschränken und folglich wird die Bearbeitung beendet und das nächste Dreieck aus *triangleQueue* gezogen.

3.2.5 Bearbeiten eines nicht sichtbaren Dreiecks

Wenn in $relevantTrianglesEdges$ dem Dreieck $t_{current}$ kein relevantes Dreieck zugeordnet ist, handelt es sich bei seinen zu v_{start} bzw. p_{start} zugeneigten Kanten um die Kanten eines Hindernispolygons. $t_{current}$ befindet sich daher vollständig hinter dem entsprechenden Hindernispolygon und ist demnach nicht sichtbar.
Wenn $t_{current}$ noch auf mindestens ein relevantes Dreieck wartet, welcher in *triangleQueue* noch zu keinem Zeitpunkt enthalten war (weiter oben erwähnter Fall), so handelt es sich bei dem relevanten Dreieck bzw. bei den relevanten Dreiecken um nicht sichtbare Dreiecke. Daher ist $t_{current}$ ebenfalls nicht sichtbar.
In beiden genannten Fällen werden in *constraintList* für alle drei Kanten $e = (v, w) \in V^2$ in $t_{current}$ die beiden Knoten v und w eingetragen, um eine Sichteinschränkung über die gesamte Kante e zu hinterlegen. Anschließend wird die Bearbeitung beendet und das nächste Dreieck aus *triangleQueue* gezogen.
In Abbildung 3.6 können die Dreiecke t_3 und t_4 nicht bearbeitet werden, solange t_5 noch nicht bearbeitet ist. Allerdings kann das Dreieck t_5 nur durch die Bearbeitung von t_3 oder t_4 in *triangleQueue* eingefügt werden. Aufgrund dieses Dillemas ist es erforderlich, das Dreieck t_4 zu bearbeiten und sein relevantes Dreieck t_5 dabei als vollständig nicht sichtbar zu betrachten, da t_5 andernfalls bereits in *triangleQueue* enthalten gewesen wäre.

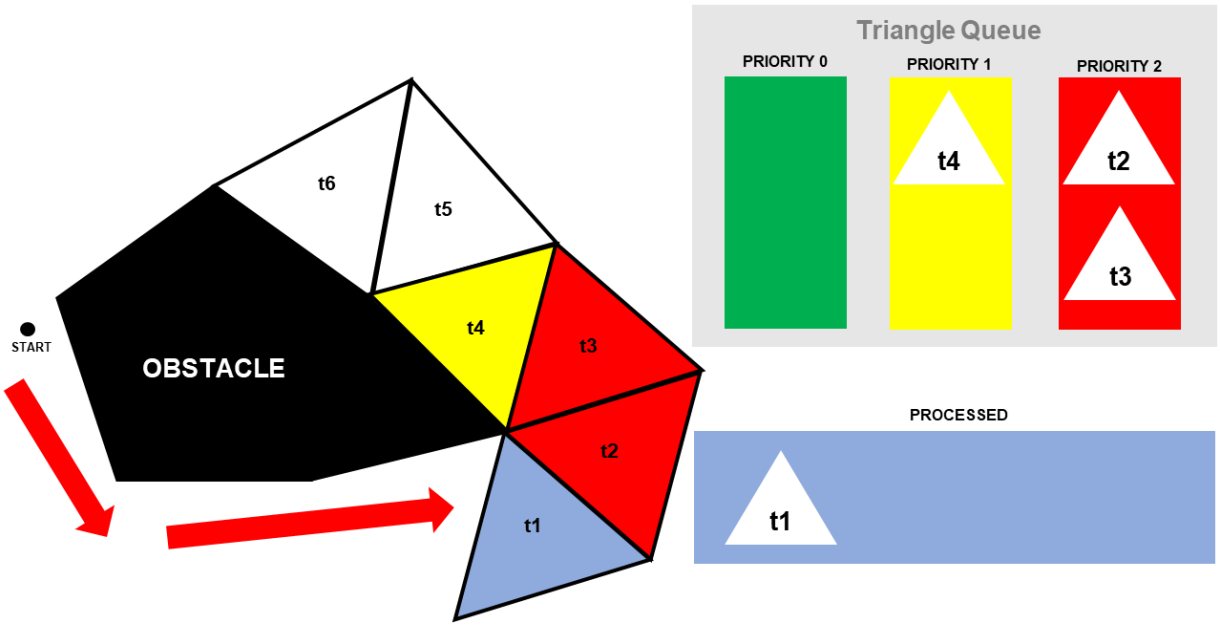


Abbildung 3.6: Nicht sichtbare Dreiecke

3.2.6 Look-Up oder Vollständige Sichteinschränkung

Der Algorithmus untersucht für jede der drei Kanten von $t_{current}$ ob sie in $relevantTrianglesEdges$ bereits als relevante Kante eines für $t_{current}$ relevanten Dreiecks $t_{relevant}$ hinterlegt worden ist. Ist dies der Fall, so wird in der Look-Up Tabelle von $constraintList$ ein entsprechender Verweis gesetzt, sodass sämtliche die Kante von $t_{current}$ betreffende Sichteinschränkungen effizient aus den Hinterlegungen für dieselbe Kante von $t_{relevant}$ abgelesen werden können. Diese Kante wird anschließend als „bearbeitet“ markiert.
Wird für eine Kante von $t_{current}$ kein Look-Up gesetzt, so untersucht der Algorithmus im Anschluss, ob es sich bei ihr um eine v_{start} bzw. p_{start} zugeneigte Kante handelt. Ist dies der Fall, so repräsentiert sie die Kante eines Hindernispolygons. Folglich werden für diese Kante in $constraintList$ ihre beiden Knoten eingetragen, um eine Sichteinschränkung über die gesamte Kante zu hinterlegen. Diese Kante wird anschließend als „bearbeitet“ markiert. In Abbildung 3.7 kann auf der linken Seite für die Kante e des Dreiecks t_2 ein Look-Up auf dieselbe Kante e des Dreiecks t_1 hinterlegt werden. Auf der rechten Seite ist für die Kante e des Dreiecks t_2 kein Look-Up auf dieselbe Kante eines anderen Dreiecks möglich, daher befindet sich die Kante e direkt hinter einem Hindernis.

3.2.7 Propagieren der hinterlegten Sichteinschränkungen

Im wichtigsten Teil des Algorithmus werden die noch nicht als „bearbeitet“ markierten Kanten $e_{current} = (v, w) \in V^2$ von $t_{current}$ betrachtet.
Als Erstes wird geschaut, ob für $t_{current}$ exakt zwei relevante Dreiecke und dementsprechend auch exakt zwei relevante Kanten vorliegen, die beide einer vollständigen Sichteinschränkung unterliegen. Ist dies der Fall, dann werden in $constraintList$ für die Kante $e_{current}$ die beiden Knoten v und w eingetragen, um eine Sichteinschränkung über die gesamte Kante zu hinterlegen. Andern-

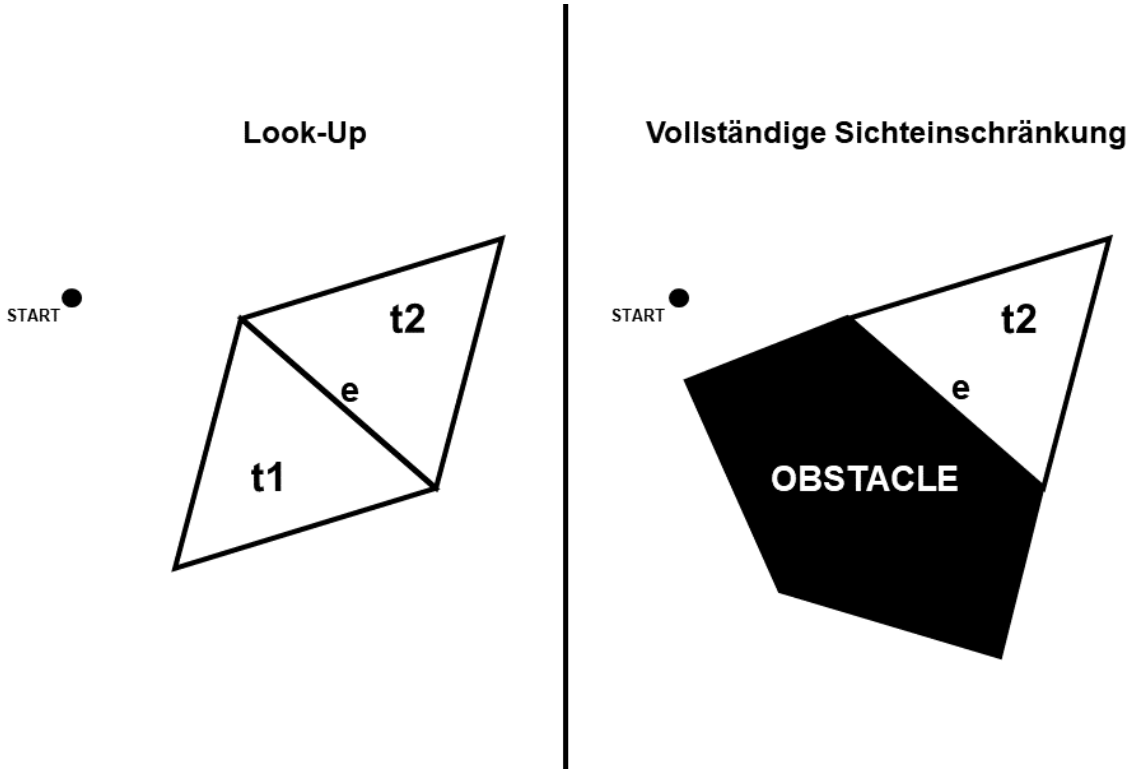


Abbildung 3.7: Look-Up oder Vollständige Sichteinschränkung

falls werden für die Bearbeitung der Kante $e_{current}$ die bereits als „bearbeitet“ markierten Kanten $e_{relevant}$ von $t_{current}$ betrachtet. Im Folgenden bezeichnet $v_{shared} \in V$ den Knoten, welchen die beiden Kanten $e_{current}$ und $e_{relevant}$ gemeinsam haben und v_{second} bezeichnet denjenigen Knoten von der Kante $e_{current}$, welcher ungleich v_{shared} ist.
Nun iteriert der Algorithmus über jede Sichteinschränkung $constraint_{relevant} = (a, b) \in V^2$, die in *constraintList* für die Kante $e_{relevant}$ hinterlegt ist.
Wenn $a = v_{shared}$ oder $b = v_{shared}$ gilt, so wird in *constraintList* für die Kante $e_{current}$ der Knoten v_{shared} hinterlegt. Dieser repräsentiert den Beginn einer Sichteinschränkung $constraint_{new}$ auf der Kante $e_{current}$. Nun wird geprüft, ob der Knoten v_{second} in dem von der Geraden definiert durch v_{start} bzw. p_{start} und a sowie der Geraden definiert durch v_{start} bzw. p_{start} und b aufgespannten Kegel liegt. Ist dies der Fall, so wird in *constraintList* für die Kante $e_{current}$ der Knoten v_{second} hinterlegt. In Abbildung 3.8 ist das beschriebene Verfahren veranschaulicht.
Liegt der Knoten v_{second} nicht in dem Kegel, wird in *constraintList* für die Kante $e_{current}$ der Knoten a (falls $b = v_{shared}$) bzw. b (falls $a = v_{shared}$) hinterlegt. In Abbildung 3.9 ist das beschriebene Verfahren veranschaulicht. In beiden Fällen repräsentiert der hinterlegte Knoten das Ende der Sichteinschränkung $constraint_{new}$ auf der Kante $e_{current}$. Die Sichteinschränkung $constraint_{relevant}$ ist abgearbeitet, sodass folglich die nächste Sichteinschränkung auf der Kante $e_{relevant}$ betrachtet werden kann.
Wenn $a \neq v_{shared}$ und $b \neq v_{shared}$ gilt, wird geschaut, ob der Knoten v_{second} in dem von der Geraden definiert durch v_{start} bzw. p_{start} und a sowie der Geraden definiert durch v_{start} bzw. p_{start} und b aufgespannten Kegel liegt.
Liegt v_{second} in dem Kegel, so wird in *constraintList* für die Kante $e_{current}$ der Knoten v_{second} hinterlegt. Dieser repräsentiert den Beginn einer Sichteinschränkung $constraint_{new}$ auf der Kante

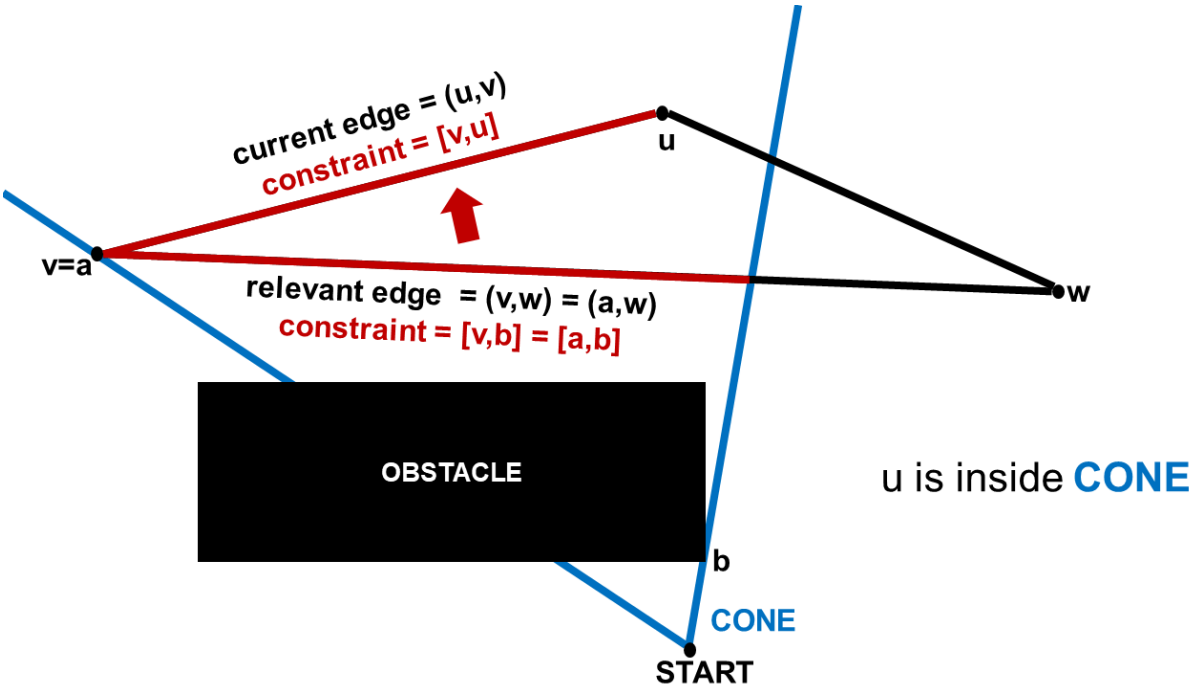


Abbildung 3.8: Beispiel für die Anwendung des beschriebenen Verfahrens zur Übertragung einer Sichteinschränkung von $e_{relevant}$ (relevant edge) auf $e_{current}$ (current edge) mit $v = v_{shared}$ und $u = v_{second}$

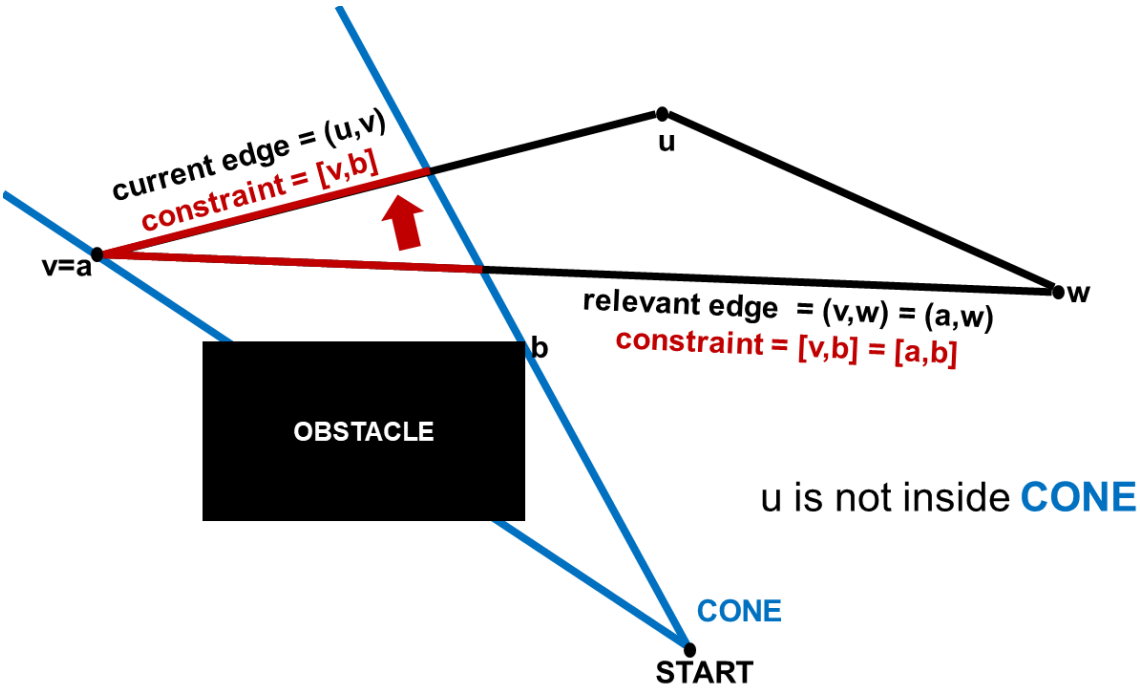


Abbildung 3.9: Beispiel für die Anwendung des beschriebenen Verfahrens zur Übertragung einer Sichteinschränkung von $e_{relevant}$ (relevant edge) auf $e_{current}$ (current edge) mit $v = v_{shared}$ und $u = v_{second}$

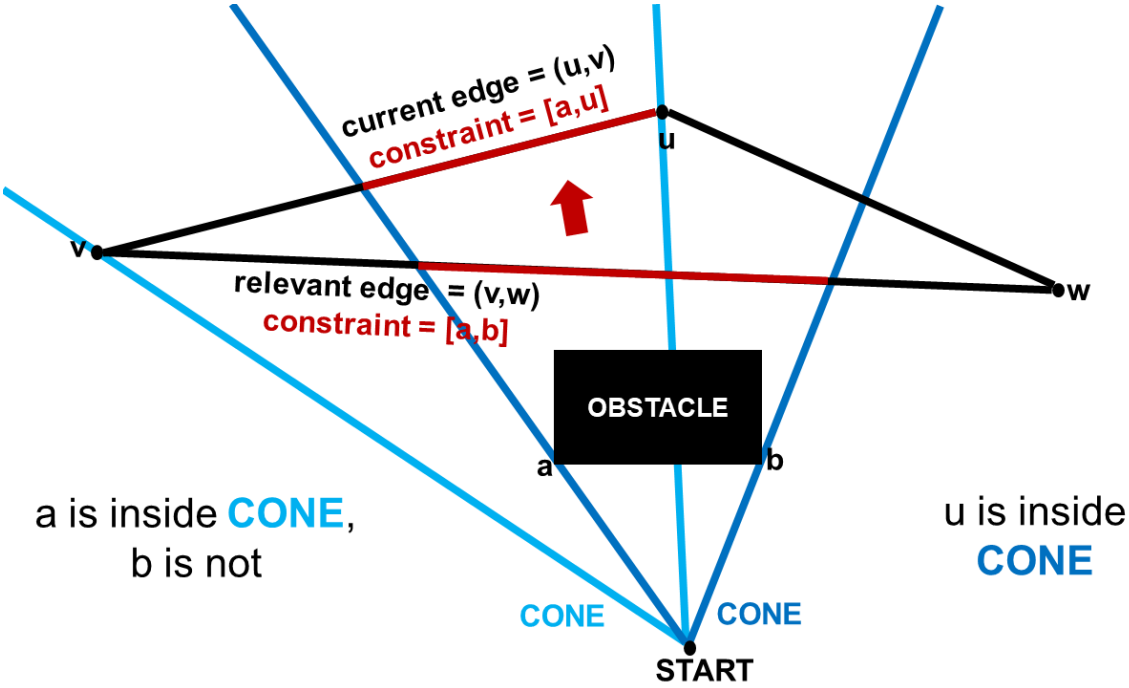


Abbildung 3.10: Beispiel für die Anwendung des beschriebenen Verfahrens zur Übertragung einer Sichteinschränkung von $e_{relevant}$ (relevant edge) auf $e_{current}$ (current edge) mit $v = v_{shared}$ und $u = v_{second}$

$e_{current}$. Nun wird geprüft, ob der Knoten a oder der Knoten b in dem von der Geraden definiert durch v_{start} bzw. p_{start} und v_{shared} sowie der Geraden definiert durch v_{start} bzw. p_{start} und v_{second} aufgespannten Kegel liegt. Derjenige von den beiden Knoten a oder b, welcher sich im Kegel befindet, wird in *constraintList* für die Kante $e_{current}$ hinterlegt und repräsentiert das Ende der Sichteinschränkung $constraint_{new}$. In Abbildung 3.10 ist das beschriebene Verfahren veranschaulicht. Die Sichteinschränkung $constraint_{relevant}$ ist abgearbeitet, sodass folglich die nächste Sichteinschränkung auf der Kante $e_{relevant}$ betrachtet werden kann.

Liegt v_{second} außerhalb dem Kegel, so wird geprüft, ob beide Endpunkte der Kante sich auf derselben Seite bezüglich dem von der Geraden definiert durch v_{start} bzw. p_{start} und a sowie der Geraden definiert durch v_{start} bzw. p_{start} und b aufgespannten Kegel befinden. Ist dies der Fall, dann besitzt die Sichteinschränkung $constraint_{relevant}$ keine Relevanz für die Kante $e_{current}$, sodass $constraint_{relevant}$ abgearbeitet ist und folglich die nächste Sichteinschränkung auf der Kante $e_{relevant}$ betrachtet werden kann. In Abbildung 3.11 ist das beschriebene Verfahren veranschaulicht.

Andernfalls werden in *constraintList* für die Kante $e_{current}$ die beiden Knoten a und b hinterlegt, da sie ein nicht sichtbares Segment auf der Kante $e_{current}$ definieren. In Abbildung 3.12 ist das beschriebene Verfahren veranschaulicht.

Die Sichteinschränkung $constraint_{relevant}$ ist abgearbeitet, sodass die nächste Sichteinschränkung auf der Kante $e_{relevant}$ betrachtet werden kann. Nachdem sämtliche Sichteinschränkungen der Kanten von $t_{current}$ übertragen worden sind, ist $t_{current}$ abgearbeitet und folglich wird das nächste Dreieck aus *triangleQueue* gezogen.

Sobald sich in *triangleQueue* keine Dreiecke mehr befinden, enthält *constraintList* sämtliche Sichteinschränkungen einer jeden Dreieckskante bis zu der dem $VertexFinder$ vorgegebenen Di-

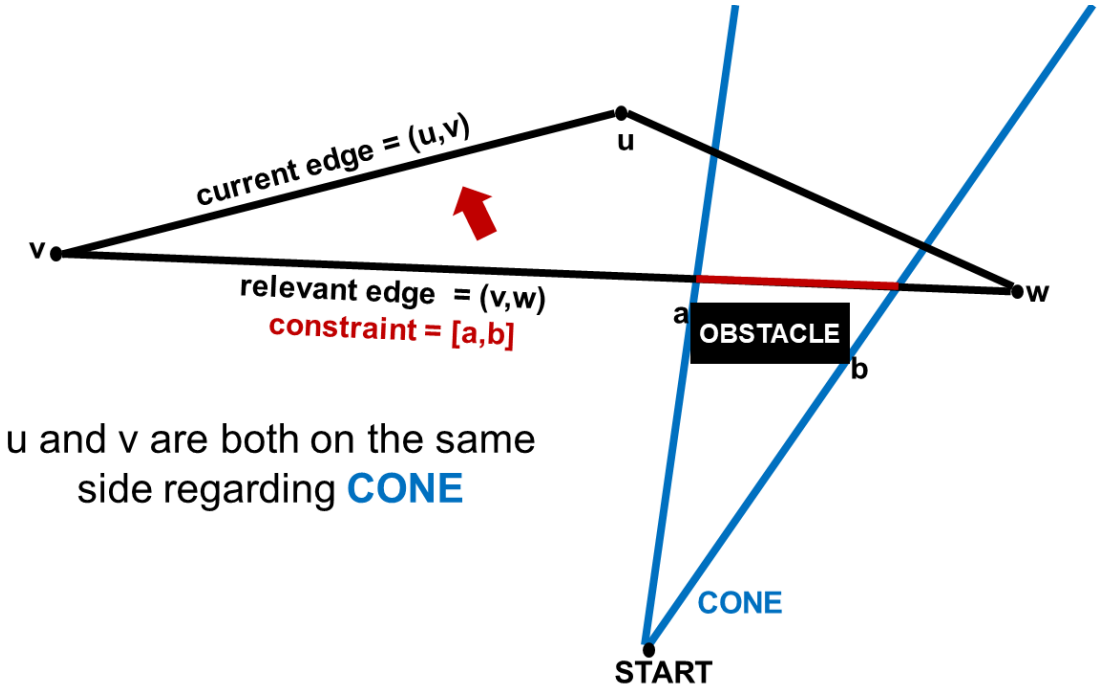


Abbildung 3.11: Beispiel für die Anwendung des beschriebenen Verfahrens zur Übertragung einer Sichteinschränkung von $e_{relevant}$ (relevant edge) auf $e_{current}$ (current edge) mit $v = v_{shared}$ und $u = v_{second}$

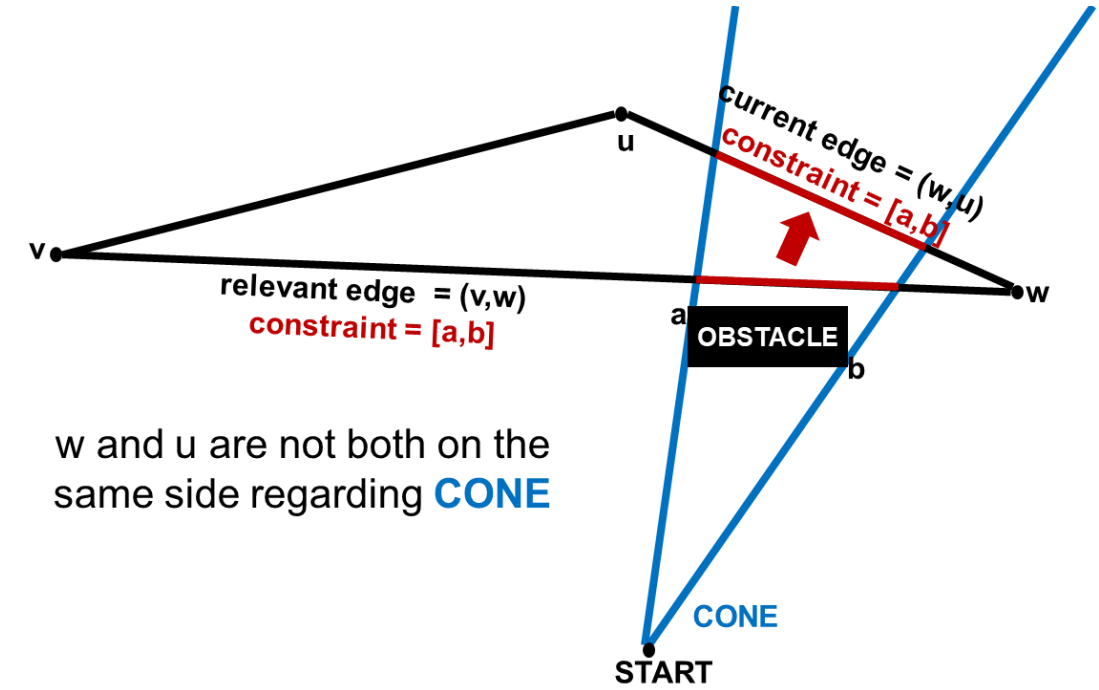


Abbildung 3.12: Beispiel für die Anwendung des beschriebenen Verfahrens zur Übertragung einer Sichteinschränkung von $e_{relevant}$ (relevant edge) auf $e_{current}$ (current edge) mit $w = v_{shared}$ und $u = v_{second}$

stanz zu v_{start} bzw. p_{start}. Mit den in *constraintList* hinterlegten Sichteinschränkungen können die von v_{start} bzw. p_{start} aus sichtbaren Hindernisecken einfach ermittelt werden.

3.2.8 Aufsammeln der sichtbaren Hindernisecken

Wie bereits erwähnt, ermöglicht *constraintList* das Aufsammeln der sichtbaren Hindernisecken. Hierzu erfolgt eine Schleife über sämtliche während der Ausführung des Algorithmus berücksichtigten Dreiecke $t \in T$. Für jeden Knoten v des Dreiecks t wird mit der Routine *isObstacle* geschaut, ob es sich bei v um eine Hindernisecke handelt. Wenn ja, wird anschließend überprüft, ob v für mindestens eine seiner adjazenten Kanten e nicht als Endpunkt einer Sichteinschränkung hinterlegt ist. Ist dies der Fall, so wird v in die Liste *vertexList* der von v_{start} bzw. p_{start} aus sichtbaren Hindernisecken aufgenommen.

3.3 Konstruktion des Sichtbarkeitsgraphen unter Verwendung des Vertex Finder

Im ersten Schritt iteriert der Algorithmus über sämtliche Knoten im Graphen $G = (V, T)$ und prüft mit der Routine *isObstacle* für jeden einzelnen Knoten, ob es sich um eine Hindernisecke handelt.
Die Hindernisecken werden in ein Array geschrieben und bilden die Knotenmenge für den Sichtbarkeitsgraphen.
Nun berechnet der $VertexFinder$ für jede Hindernisecke $v \in V$ alle von ihm aus sichtbaren Hindernisecken und für jede sichtbare Hindernisecke $w \in V$ wird im zu konstruierenden Sichtbarkeitsgraphen eine Kante von v nach w hinterlegt.

4 Dijkstra-basierter Algorithmus

In diesem Kapitel wird zunächst die iterative Anwendung des A* Algorithmus zur Berechnung des kürzesten Pfades und anschließend die Umsetzung des A* Algorithmus unter Verwendung des $VertexFinder$ erläutert.

4.1 Berechnung des kürzesten Pfades mit iterativem A*

Zur Berechnung des kürzesten Pfades von einem Punkt p_{start} zu einem Punkt p_{target} wird zunächst eine obere Schranke für die Länge einer Kante auf dem kürzesten Pfad ermittelt. Dieses beträgt die Summe aus der Differenz der x-Koordinaten der beiden Hindernisecken mit niedrigster und höchster x-Koordinate und der Differenz der y-Koordinaten der beiden Hindernisecken mit niedrigster und höchster y-Koordinate. d_{max} definiert die dem $VertexFinder$ im weiteren Verlauf vorgegebene Distanz (also jene Distanz bis zu der sämtliche von einem Knoten oder einem Punkt aus sichtbaren Hindernisecken ermittelt werden). Zu Beginn wird d_{max} auf die euklidische Distanz zwischen p_{start} und p_{target} gesetzt. Nun beginnt eine Schleife, welche mit dem Auffinden des kürzesten Pfades oder einer Feststellung der Nichtexistenz eines Pfades von p_{start} nach p_{target} terminiert.
Zu Beginn einer jeden Iteration wird der Wert von d_{max} verdoppelt. Nun wird mit der im zweiten Abschnitt dieses Kapitels erläuterten Routine ein Array berechnet, das einen Pfad π von p_{start} nach p_{target} repräsentiert, welcher unter Verwendung von d_{max} als die dem $VertexFinder$ vorgegebene Distanz optimal ist.
Da im zurückgegebenen Array für jede Hindernisecke $w \in V$ und für den Punkt p_{target} die vorherige sichtbare Hindernisecke $v \in V$ bzw. der sichtbare Punkt p_{start} auf dem kürzesten Pfad von p_{start} nach w bzw. nach p_{target} eingetragen ist, kann der kürzeste Pfad von p_{start} nach p_{target} sowie seine Distanz einfach berechnet werden, indem von p_{target} aus beginnend, sämtliche auf dem kürzesten Pfad liegende Hindernisecken in entgegengesetzter Reihenfolge ermittelt werden, bis der Punkt p_{start} erreicht ist.
Mit dem im dritten Abschnitt des zweiten Kapitels sowie eben beschriebenen Verfahren wird der Pfad π im Anschluss konstruiert und seine Gesamtdistanz berechnet. Liegt die Gesamtdistanz höchstens bei dem Wert von d_{max}, dann ist der kürzeste Pfad π von p_{start} nach p_{target} gefunden und der Algorithmus wird beendet. Andernfalls wird eine neue Iteration gestartet, in welcher der Wert von d_{max} erneut verdoppelt wird. Hatte der Wert von d_{max} zuvor bereits die obere Schranke für die Länge einer Kante auf dem kürzesten Pfad erreicht, dann existiert kein Pfad π von p_{start} nach p_{target} und der Algorithmus wird beendet.

4.2 Anwendung des Vertex Finder zur Umsetzung des A* Algorithmus

Für diesen Algorithmus nehmen wir den Wert von d_{max} als vorgegeben an. Es soll in einem euklidischen Raum mit Hindernissen der unter Verwendung von d_{max} kürzeste Pfad von einem

Punkt p_{start} zu einem Punkt p_{target} berechnet werden. Der folgende Pseudo-Code beschreibt den hierzu entwickelten Algorithmus:

Algorithm 3 A*$(G, p_{start}, p_{target})$

```
for v ∈ V do
  distances[v] ← ∞
end for
if isVisible(p_start, p_target) then
  previousShortestPathVertex[p_target] ← p_start
  return previousShortestPathVertex
end if
visibleObstacleVertices ← vertexFinder.findVisibleObstacleVertices(p_start, d_max)
for w ∈ visibleObstacleVertices do
  distances[w] ← euclDist(p_start, w)
  modifiedDistances[w] ← distances[w] + euclDist(w, p_target)
  priorityQueue.insertOrUpdate(w, modifiedDistances[w])
  previousShortestPathVertex[w] ← p_start
end for
while NOT priorityQueue.isEmpty() do
  v ← priorityQueue.popMin()
  if isVisible(v, p_target) then
    previousShortestPathVertex[p_target] ← v
    return previousShortestPathVertex
  end if
  visibleObstacleVertices ← vertexFinder.findVisibleObstacleVertices(v, d_max)
  for w ∈ visibleObstacleVertices do
    if distances[v] + euclDist(v, w) < distances[w] then
      distances[w] ← distances[v] + euclDist(v, w)
      modifiedDistances[w] ← distances[w] + euclDist(w, p_target)
      priorityQueue.insertOrUpdate(w, modifiedDistances[w])
      previousShortestPathVertex[w] ← v
    end if
  end for
end while
return null =0
```

5 Experimente und Ergebnisse

Die in den vorangegangenen Kapiteln erläuterten Algorithmen wurden mit der Programmiersprache Java implementiert und unter Ubuntu Linux ausgeführt. Der hierfür eingesetzte Rechner mit dem Modelnamen Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz besitzt einen Arbeitsspeicher von 64 GB. Bei dem Graphen handelt es sich um eine Darstellung des ägäischen Meeres östlich von Kreta mit den Längengraden (hier x-Koordinaten) zwischen 22.0 und 27.1 sowie den Breitengraden (hier y-Koordinaten) zwischen 34.0 und 37.56. Der Graph enthält insgesamt $277,096$ Knoten und $458,153$ Dreiecke. Unter den Knoten finden sich insgesamt $104,954$ Hindernisecken wieder.

Ausführen des Programms
Der einzulesende Originalgraph muss sich in einer .graph-Datei mit dem Namen *input* in dem gleichen Verzeichnis wie das auszuführende Programm befinden.
Der einzulesende Sichtbarkeitsgraph muss sich in einer .graph-Datei mit dem Namen *visibility* in dem gleichen Verzeichnis wie das auszuführende Programm befinden.
Wenn zum Start des Programms keine Übergabeparameter vorliegen, wird der $VertexFinder$ einmal für den bereits im Programm definierten Startknoten mit dem Index 7837 und einmal für den bereits im Programm definierten Startpunkt mit den Koordinaten $(24.5; 36)$ ausgeführt.
Es ist auch möglich, Parameter zu übergeben.
Der erste Übergabeparameter muss eine Zahl aus $\{1, 2, 3, 4\}$ sein. Damit kann der auszuführende Algorithmus folgendermaßen gewählt werden:
„1“ $\implies$ Naiver $VertexFinder$
„2“ $\implies$ $VertexFinder$
„3“ $\implies$ Dijkstra auf dem Sichtbarkeitsgraphen
„4“ $\implies$ Dijkstra unter Verwendung des $VertexFinder$
Der zweite Übergabeparameter muss eine Fließkommazahl im Intervall zwischen 22.0 und 27.1 sein. Damit wird die x-Koordinate des Startpunkts bestimmt.
Der dritte Übergabeparameter muss eine Fließkommazahl im Intervall zwischen 34.0 und 37.56 sein. Damit wird die y-Koordinate des Startpunkts bestimmt.
Sofern es sich bei dem ersten Übergabeparameter um eine Zahl aus $\{3, 4\}$ handelt, müssen noch zwei weitere Parameter übergeben werden.
Der vierte Übergabeparameter muss eine Fließkommazahl im Intervall zwischen 22.0 und 27.1 sein. Damit wird die x-Koordinate des Zielpunkts bestimmt.
Der fünfte Übergabeparameter muss eine Fließkommazahl im Intervall zwischen 34.0 und 37.56 sein. Damit wird die y-Koordinate des Zielpunkts bestimmt.
Als Ausgabe wird eine .gl-Datei in dem gleichen Verzeichnis wie das auszuführende Programm erstellt, die mit einem geeigneten 3D-Viewer visualisiert werden kann. In ihrem Dateinamen sind bestimme Metadaten über die Ausführung des Algorithmus enthalten.

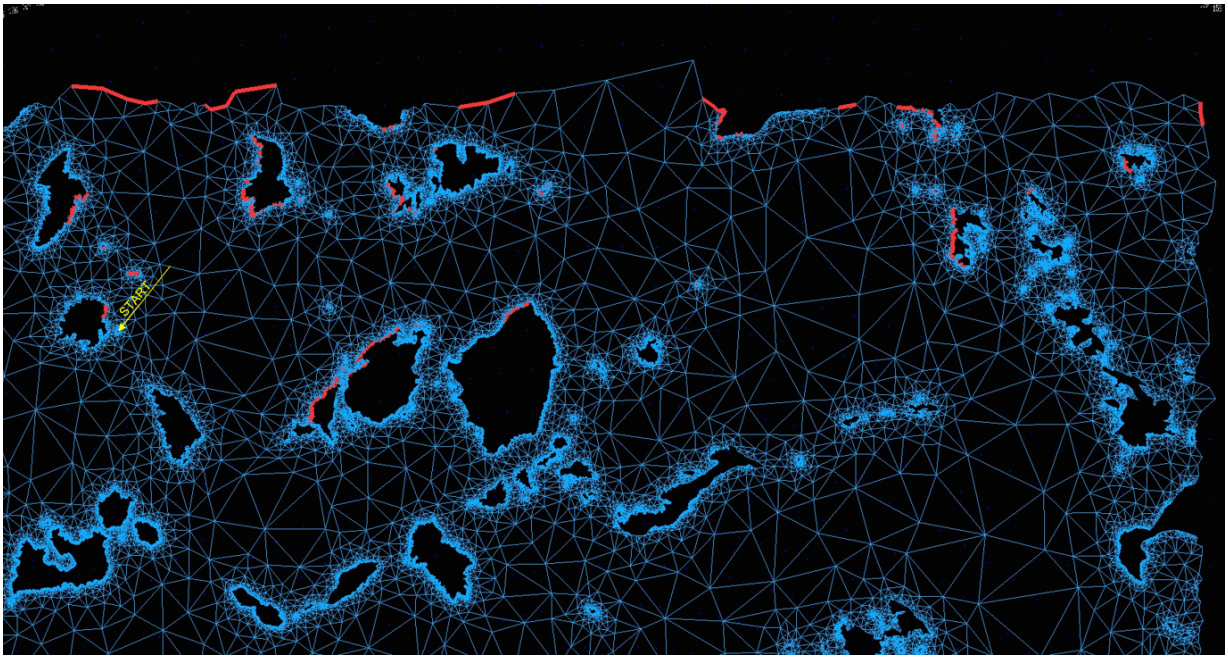


Abbildung 5.1: Visualisierung der Ausgabe vom naiven $VertexFinder$. In Rot sind die Kanten der vom Startknoten mit dem Index 7837 aus sichtbaren Hindernisecken markiert.

5.1 Naiver Vertex Finder

Der naive $VertexFinder$ berechnet für einen gegebenen Startknoten $v_{start} \in V$ oder für einen gegebenen Startpunkt p_{start} mit seinen Koordinaten $(p_{start}.x, p_{start}.y)$ bis zu einer vorgegebenen Distanz alle Hindernisecken die von v_{start} bzw. p_{start} aus sichtbar sind. Hierzu prüft der Algorithmus für jeden Knoten $v \in V$, ob dieser sich innerhalb der vorgegebenen Distanz zu v_{start} bzw. p_{start} befindet. Anschließend wird mit der Routine $isObstacle$ geschaut, ob es sich bei v um eine Hindernisecke handelt. Wenn beides zutrifft, wird mit der Routine $isVisible$ untersucht, ob der Knoten v von v_{start} bzw. p_{start} aus sichtbar ist. Ist dies der Fall, wird v in die Liste $vertexList$ der von v_{start} bzw. p_{start} aus sichtbaren Hindernisecken aufgenommen. Hierbei werden Dreiecke mehrfach bearbeitet, da die Routine $isVisible$ für jede einzelne Hindernisecke $v \in V$ sämtliche zwischen v_{start} bzw. p_{start} und v liegende Dreiecke untersucht, sodass die Gesamtzahl an Bearbeitungen für n Dreiecke in $\theta(n^2)$ liegen kann.
Für 1000 zufällig gewählte Punkte benötigt der Algorithmus im Durchschnitt 3.1635 Sekunden. In Abbildung 5.1 ist die Ausgabe des naiven $VertexFinder$ für den Startknoten mit dem Index 7837 und in Abbildung 5.2 ist die Ausgabe des naiven $VertexFinder$ für den Startpunkt mit den Koordinaten $(24.5; 36)$ visualisiert.

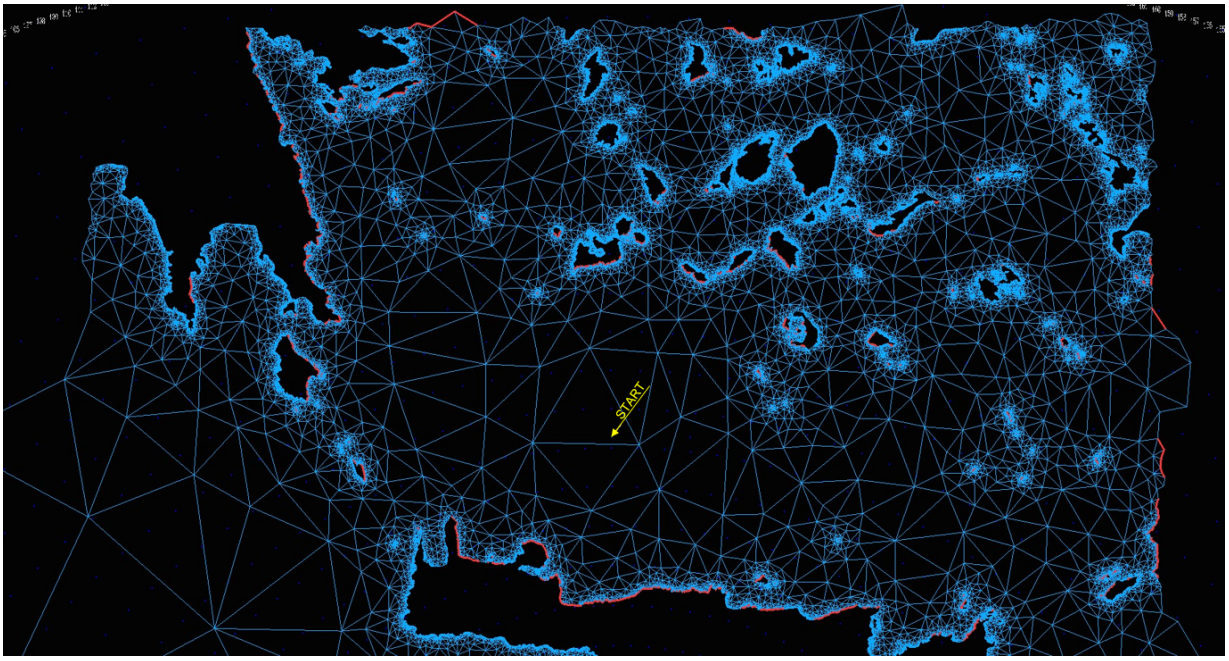


Abbildung 5.2: Visualisierung der Ausgabe vom naiven $VertexFinder$. In Rot sind die Kanten der vom Startpunkt mit den Koordinaten $(24.5; 36)$ aus sichtbaren Hindernisecken markiert.

5.2 Vertex Finder

Der $VertexFinder$ berechnet mit dem in Kapitel 3 beschriebenen Algorithmus für einen gegebenen Startknoten $v_{start} \in V$ oder für einen gegebenen Startpunkt p_{start} mit seinen Koordinaten $(p_{start}.x, p_{start}.y)$ bis zu einer vorgegebenen Distanz alle Hindernisecken die von v_{start} bzw. p_{start} aus sichtbar sind.
Für die gleichen 1000 zufällig gewählten Punkte wie im vorigen Abschnitt benötigt der Algorithmus im Durchschnitt 0.3708 Sekunden. In Abbildung 5.3 ist die Ausgabe des $VertexFinder$ für den Startknoten mit dem Index 7837 visualisiert. Hierbei bleiben $16,233$ Dreiecke ohne Sichteinschränkungen, $7,383$ Dreiecke sind teils mit Sichteinschränkungen belegt und $434,537$ Dreiecke sind vollständig nicht sichtbar. In Abbildung 5.4 ist die Ausgabe des $VertexFinder$ für den Startpunkt mit den Koordinaten $(24.5; 36)$ visualisiert. Hierbei bleiben $69,178$ Dreiecke ohne Sichteinschränkungen, $27,570$ Dreiecke sind teils mit Sichteinschränkungen belegt und $361,405$ Dreiecke sind vollständig nicht sichtbar.

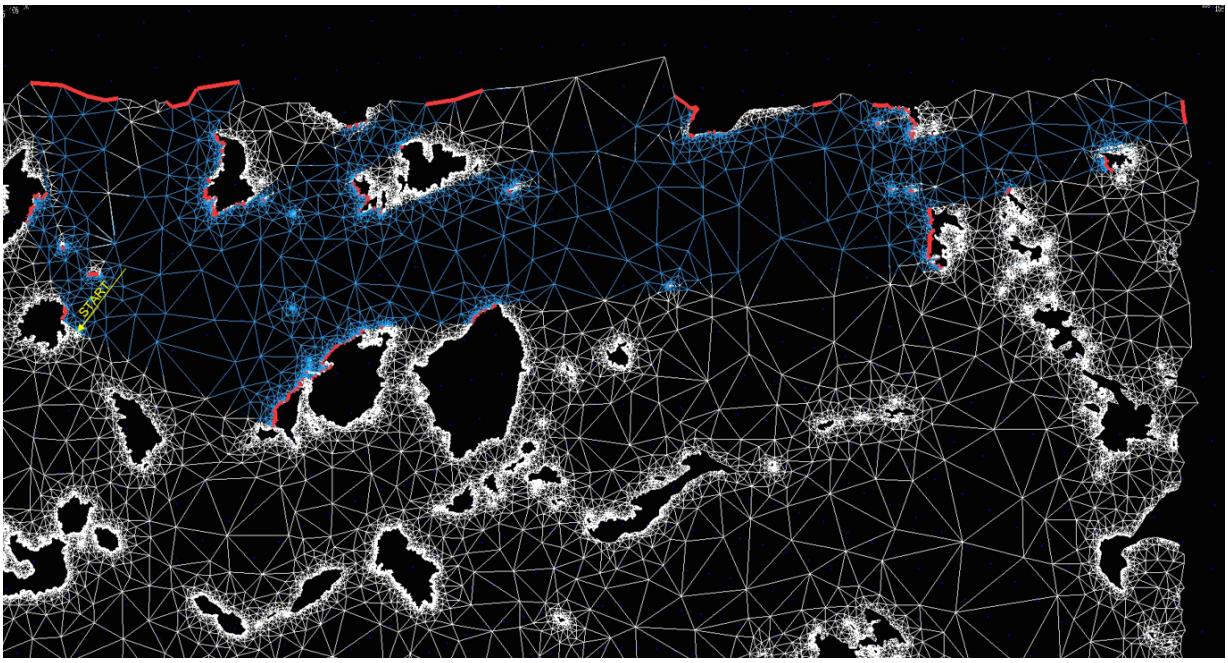


Abbildung 5.3: Visualisierung der Ausgabe vom *VertexFinder*. In Weiß sind die Sichteinschränkungen, in Rot die Kanten der vom Startknoten mit dem Index 7837 aus sichtbaren Hindernisecken markiert.

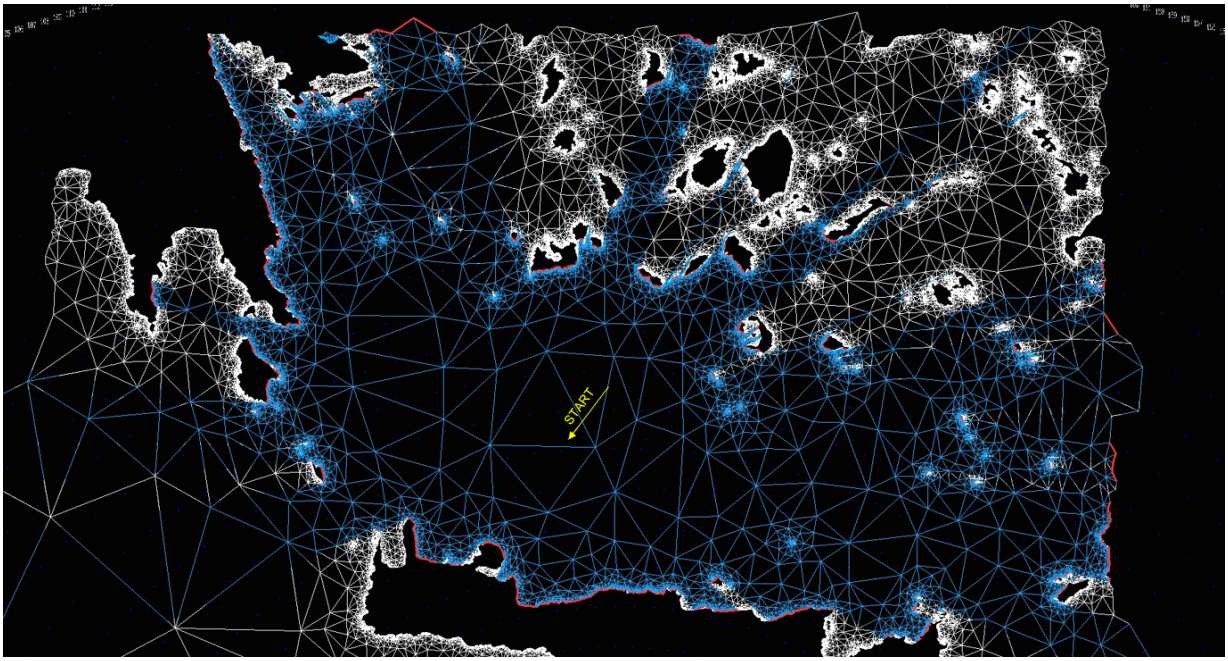


Abbildung 5.4: Visualisierung der Ausgabe vom *VertexFinder*. In Weiß sind die Sichteinschränkungen, in Rot die Kanten der vom Startpunkt mit den Koordinaten $(24.5; 36)$ aus sichtbaren Hindernisecken markiert.

5.3 Dijkstra auf dem vorberechneten Sichtbarkeitsgraphen

Der Dijkstra berechnet mit dem im dritten Abschnitt des zweiten Kapitels beschriebenen Algorithmus unter Zuhilfenahme eines vorberechneten Sichtbarkeitsgraphen den kürzesten Pfad von einem Punkt p_{start} zu einem Punkt p_{target}.
Für 10 zufällig gewählte Start- und Zielpunkte benötigt der Algorithmus im Durchschnitt 2.9804 Sekunden. In Abbildung 5.5 ist die Ausgabe des Dijkstra Algorithmus für den Startknoten mit dem Index 13652 und den Zielknoten mit dem Index 91282 visualisiert. Hierbei werden $7,090$ Hindernisecken aus der Priority Queue gezogen und insgesamt $2,542,721$ Kanten verarbeitet. In Abbildung 5.6 ist die Ausgabe des Dijkstra Algorithmus für den Startpunkt mit den Koordinaten $(24.5; 34.2)$ und den Zielpunkt mit den Koordinaten $(24.5; 37.4)$ visualisiert. Hierbei werden $6,734$ Hindernisecken aus der Priority Queue gezogen und insgesamt $2,122,931$ Kanten verarbeitet.

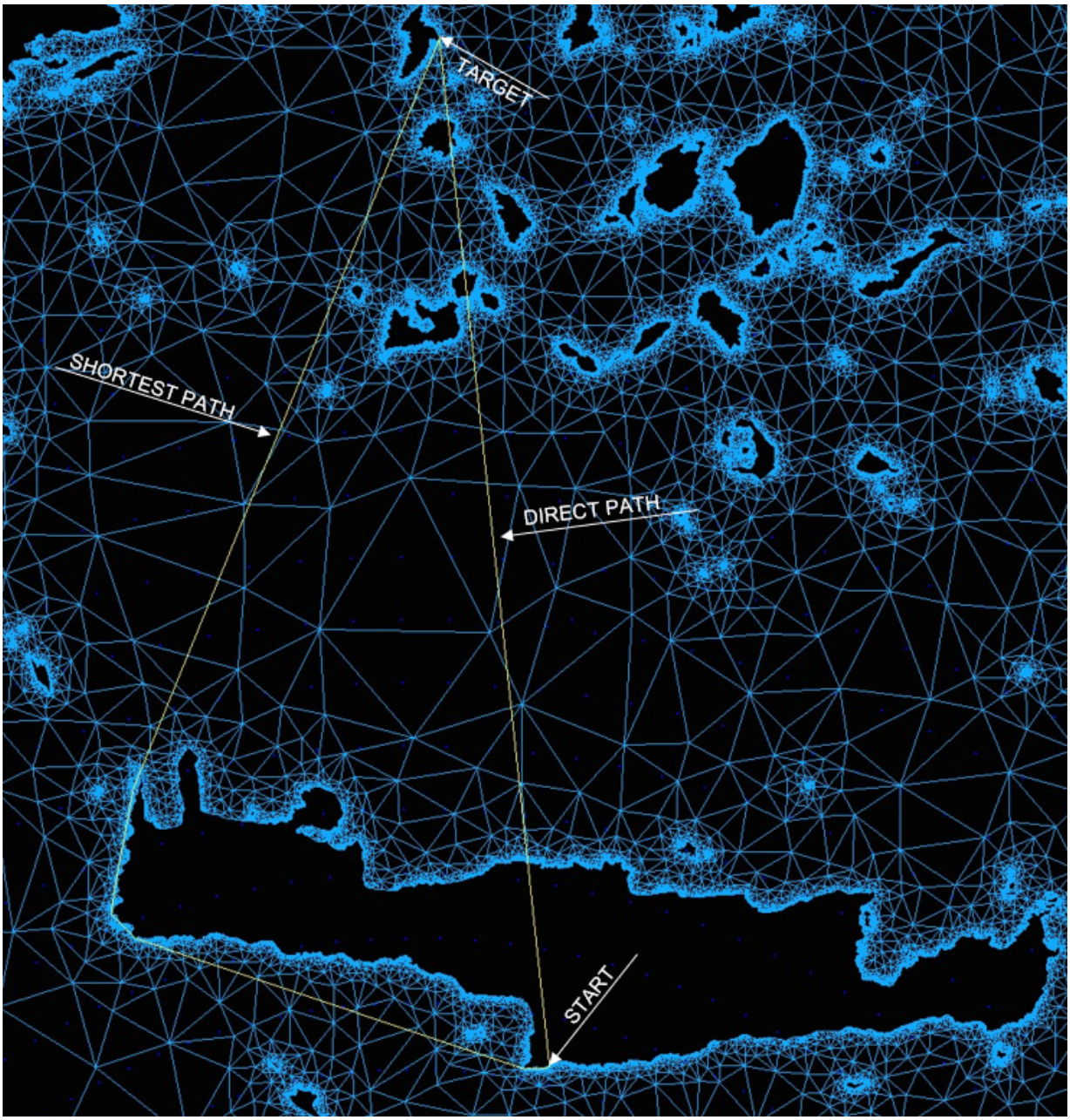


Abbildung 5.5: Visualisierung der Ausgabe vom Dijkstra Algorithmus für den Startknoten mit dem Index 13652 und den Zielknoten mit dem Index 91282

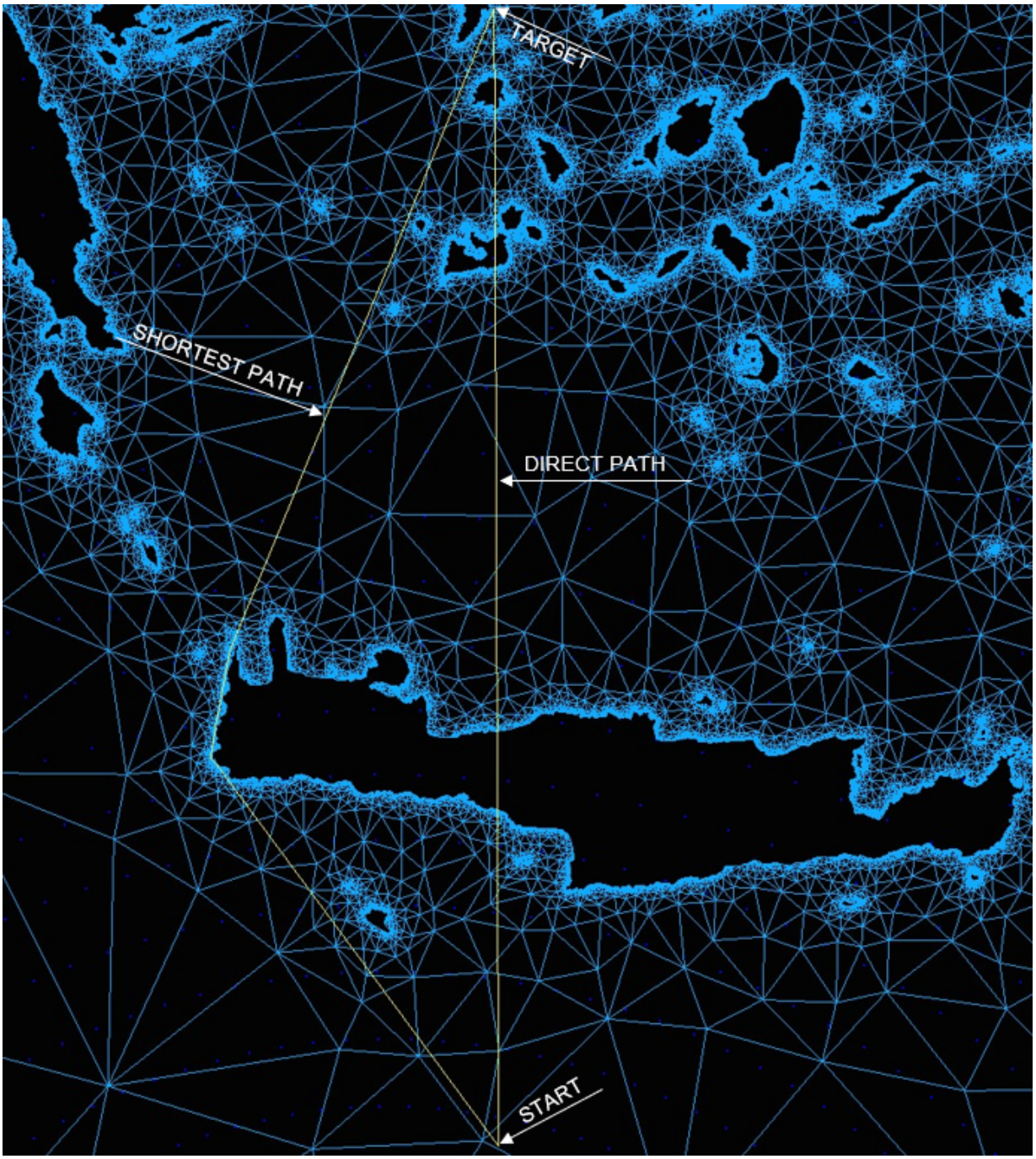


Abbildung 5.6: Visualisierung der Ausgabe vom Dijkstra Algorithmus für den Startpunkt mit den Koordinaten $(24.5; 34.2)$ und den Zielpunkt mit den Koordinaten $(24.5; 37.4)$

Algorithmus	Laufzeit (in Sekunden)
Naiver $VertexFinder$	3.1635
$VertexFinder$	0.3708
Dijkstra auf dem Sichtbarkeitsgraphen	2.9804
Dijkstra unter Verwendung des $VertexFinder$	1,729.8524
Naive Berechnung des Sichtbarkeitsgraphen	227,229.683
Berechnung des Sichtbarkeitsgraphen mit $VertexFinder$	39,990.391

Tabelle 5.1: Durchschnittliche Laufzeiten der Algorithmen

Fragment	Laufzeit (in Milisekunden)
Vorbereitung	26
Erste Schritte	56
Aufsammeln der adjazenten Dreiecke	152
Bearbeiten eines Startdreiecks	8
Bearbeiten eines nicht sichtbaren Dreiecks	10
Look-Up oder vollständige Sichteinschränkung	57
Propagieren der hinterlegten Sichteinschränkungen	56
Aufsammeln der sichtbaren Hindernisecken	94

Tabelle 5.2: Durchschnittliche Laufzeiten der einzelnen Fragmente des $VertexFinder$

5.4 Dijkstra unter Verwendung des Vertex Finder

Der Dijkstra berechnet mit dem in Kapitel 4 beschriebenen Algorithmus ohne Zuhilfenahme eines vorberechneten Sichtbarkeitsgraphen den kürzesten Pfad von einem Punkt p_{start} zu einem Punkt p_{target}.
Für die gleichen 10 zufällig gewählten Start- und Zielpunkte wie im vorigen Abschnitt benötigt der Algorithmus im Durchschnitt in etwa 28 Minuten und 50 Sekunden. Da die hierbei entstandenen Ausgaben denen aus dem vorigen Abschnitt entsprechen, wird weiterhin auf die Abbildungen 5.5 und 5.6 verwiesen.

5.5 Konstruktion des Sichtbarkeitsgraphen

Die Konstruktion des Sichtbarkeitsgraphen unter Verwendung des $VertexFinder$ benötigt in etwa 11 Stunden und 7 Minuten. Die naive Konstruktion hingegen benötigt in etwa 63 Stunden und 7 Minuten. Der Sichtbarkeitsgraph enthält insgesamt 104,954 Hindernisecken und 154,625,753 Kanten.

5.6 Vergleich und Analyse

In Tabelle 5.1 sind die durchschnittlichen Laufzeiten der Algorithmen noch einmal aufgelistet.
In Tabelle 5.2 sind die durchschnittlichen Laufzeiten der einzelnen Fragmente des $VertexFinder$ (Unterabschnitte im zweiten Abschnitt des dritten Kapitels) aufgelistet. Für diejenigen Fragmente, welche im $VertexFinder$ mehrfach durchlaufen werden, sind die benötigten Laufzeiten über sämtliche Durchläufe einer Ausführung des $VertexFinder$ aufsummiert worden.
In Tabelle 5.3 sind die Anzahlen der Dreiecke ohne Sichteinschränkung, mit teilweiser Sichtein-

Anzahl Dreiecke...	Startknoten 7837	Startpunkt (24.5; 36)
... ohne Sichteinschränkung	16,233	69,178
... mit teilweiser Sichteinschränkung	7,383	27,570
... mit vollständiger Sichteinschränkung	434,537	361,405

Tabelle 5.3: Anzahl Dreiecke ohne / mit teilweiser / mit vollständiger Sichteinschränkung nach Ausführung des $VertexFinder$

Anzahl bearbeiteter...	... Hindernisecken	... Kanten
Startknoten 13652, Zielknoten 91282	7,090	2,542,721
Startpunkt (24.5; 34.2), Zielpunkt (24.5; 37.4)	6,734	2,122,931

Tabelle 5.4: Anzahl bearbeiteter Hindernisecken und Anzahl bearbeiteter Kanten nach Ausführung des Dijkstra Algorithmus auf dem Sichtbarkeitsgraphen

schränkung und mit vollständiger Sichteinschränkung nach Ausführung des $VertexFinder$ einmal für den Startknoten mit dem Index 7837 und einmal für den Startpunkt mit den Koordinaten (24.5; 36) aufgelistet.

In Tabelle 5.4 sind die Anzahlen der bearbeiteten Hindernisecken und der bearbeiteten Kanten nach Ausführung des Dijkstra Algorithmus auf dem Sichtbarkeitsgraphen für den Startknoten mit dem Index 13652 und den Zielknoten mit dem Index 91282 sowie für den Startpunkt mit den Koordinaten (24.5; 34.2) und den Zielpunkt mit den Koordinaten (24.5; 37.4) aufgelistet.

Aus den Experimenten geht hervor, dass der $VertexFinder$ weniger als $\frac{1}{8}$ der Laufzeit des naiven $VertexFinder$ benötigt. Komplexitätstechnisch wird beim $VertexFinder$ jedes Dreieck nur einmal bearbeitet, während beim naiven $VertexFinder$ ein Dreieck u.U. $\theta(n)$ mal untersucht werden muss. Das erklärt auch, weshalb die naive Konstruktion des Sichtbarkeitsgraphen mit einer Laufzeit von ca. 63 Stunden etwa 6 mal so aufwändig ist, wie die Konstruktion des Sichtbarkeitsgraphen unter Verwendung des $VertexFinder$, welche eine Laufzeit von ca. 11 Stunden benötigt. Darüber hinaus ist nach Ausführung des $VertexFinder$ eine große Anzahl an Dreiecken vollständig nicht sichtbar, sogar im Falle eines Startpunkts welcher sich nicht in der Nähe eines Hindernispolygons befindet. Dreiecke in der Nähe eines Hindernispolygons sind im Allgemeinen wesentlich kleiner, als jene im freien Raum, daher liegen tendentiell mehr Dreiecke in der Nähe eines Hindernispolygons und folglich auch mehr Dreiecke im nicht sichbaren Bereich, unabhänig davon wie der Startpunkt gewählt wurde. Der Dijkstra Algorithmus auf dem Sichtbarkeitsgraphen hat eine Laufzeit von nur wenigen Sekunden. Bei dem hier gewählten A*-Algortihmus werden nur wenige Hindernisecken bearbeitet, wobei die meisten Hindernisecken einen sehr hohen Grad besitzen. Allerdings setzt seine Anwendung das Vorhandensein eines Sichtbarkeitsgraphen voraus, dessen Vorberechnung sehr aufwändig und dessen Speicherung ab einer bestimmten Größe des originalen Graphen nicht zu bewerkstelligen ist. Der in Kapitel 4 vorgestellte Dijkstra-basierte Algorithmus hat in den Experimenten eine durchschnittliche Laufzeit von etwa 28 Minuten und 50 Sekunden. Der große Vorteil an diesem Verfahren besteht darin, dass es ohne das Vorhandensein eines vorberechneten Sichtbarkeitsgraphen anwendbar ist.

6 Fazit und Ausblick

In dieser Arbeit wurde ein Algorithmus zur effizienten Berechnung der sichtbaren Hindernisecken bis zu einer vorgegebenen Distanz von einem Startknoten bzw. Startpunkt aus vorgestellt. Für die Berechnung der sichtbaren Hindernisecken sowie für die Konstruktion des Sichtbarkeitsgraphen konnte durch den Einsatz des $VertexFinder$ gegenüber den naiven Verfahren ein deutlicher Speed-Up erzielt werden. Darüber hinaus ermöglicht dieser Algorithmus in einem daran anschließend vorgestellten Dijkstra-basierten Ansatz die Berechnung des kürzesten Pfades von einem gegebenen Startpunkt zu einem gegebenen Zielpunkt, ohne das Vorhandensein eines vorberechneten Sichtbarkeitsgraphen. Eine effizientere Implementierung des $VertexFinder$ könnte den Speed-Up sowie seine Überlegenheit gegenüber den naiven Verfahren noch weiter verstärken. Dreiecke, welche aufgrund ihrer ungünstigen Lage „offensichtlich“ in einer nicht sichtbaren Umgebung liegen, könnten während der Ausführung des $VertexFinder$ mit ausgeklügelten Techniken erkannt und vollständig ignoriert werden, sodass in einigen Fällen ein Großteil der Dreiecke im Graphen überhaupt nicht bearbeitet wird. Der Einsatz von Pruning- und Speed-Up-Techniken bietet daher ein umfangreiches Feld für weitere Forschungsaktivitäten.

Literaturverzeichnis

[1] John F. Canny and John H. Reif. New lower bound techniques for robot motion planning problems. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 49–60. IEEE Computer Society, 1987.

[2] Subir Kumar Ghosh and David M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20(5):888–910, 1991.

[3] John Hershberger. Geometric shortest paths in the plane. In *Encyclopedia of Algorithms*, pages 840–846. 2016.

[4] John Hershberger and Subhash Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM J. Comput.*, 28(6):2215–2256, 1999.

[5] Joseph S. B. Mitchell. L_1 shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8(1):55–88, 1992.

[6] Joseph S. B. Mitchell. Shortest paths among obstacles in the plane. In Chee Yap, editor, *Proceedings of the Ninth Annual Symposium on Computational GeometrySan Diego, CA, USA, May 19-21, 1993*, pages 308–317. ACM, 1993.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Druck-Exemplaren überein.

Stuttgart, den 15. Dezember 2023