Diplomarbeit zum Thema:

„Minimum Risk Routing“

zur Erlangung des Grades eines Diplom-Informatikers (FH)

Bearbeiter: Patrick Franz
Geburtsdatum: 23. Juni 1984
Geburtsort: Waiblingen
Ausbildungsstelle: Hochschule Weingarten (FH)
Studiengang: Angewandte Informatik
Matrikelnummer: 15283

Referent: Prof. Dr. rer. nat. Wolfgang Ertel
Prüfungsort: Weingarten
Abgabedatum: 28. Februar 2007

Inhalt

Abbildungsverzeichnis

Abkürzungs- und Symbolverzeichnis

Abkürzungen:

$\varnothing$	leere Menge
$\Rightarrow$	Implikation (wenn …, dann …)
$\rightarrow$	Konversion (… wird zu …)
$\subset$	echte Teilmenge
$[a,b]$	abgeschlossenes Intervall von a nach b
$[a,b[$	halboffenes Intervall von a nach b
$\lvert M \rvert$	Kardinalität einer Menge M

Symbole der Graphentheorie:

$v \in V$	ein Knoten v aus der Menge der Knoten V
$e \in E$	eine Kante e aus der Menge der Kanten E
$E \mid V'$	auf V' induzierte Kantenmenge
$e = \{v_1, v_2\}$	eine Kante e zwischen den Knoten v_1 und v_2
$e = (v_1, v_2)$	eine gerichtete Kante e zwischen v_1 und v_2
$d_{in}(v)$	Eingangsgrad (*indegree*) des Knoten v
$d_{out}(v)$	Ausgangsgrad (*outdegree*) des Knoten v
$w(e) = w(a,b) = w_{ab}$	Gewicht der Kante $e = \{a,b\}$
$G\pi = (V\pi, E\pi)$	Vorgängerteilgraph
$d[v]$	Schätzung des kürzesten Pfades
$\delta(u,v)$	Gewicht des kürzesten Pfades

1. Einführung

1.1 Der Minimum Risk Router

Ziel des Minimum Risk Routings ist die Berechnung eines kosten-optimalen Flugpfades zwischen zwei Wegpunkten innerhalb eines gegebenen operationellen Gebiets. Die Kosten des Flugpfades bestimmen sich dabei aus dem Grad der Sichtbarkeit im Gelände und aus dem Bedrohungspotential von durchflogenen Threats und Weglänge und daraus resultierenden Treibstoffverbrauch.

Der Minimum Risk Router dieser Diplomarbeit wird vor ein dreidimensionales Problem gestellt. Er soll eine günstige Route durch das Gelände finden, deren Routenpunkte sowohl durch die geographische Länge und Breite als auch durch die empfohlenen Flughöhen über Grund beschrieben werden. Die Suche wird mit dem linearen Bellman-Verfahren realisiert. Grundlage ist die Diplomarbeit von Volker Subat [3].

Das Hauptaugenmerk dieser Diplomarbeit liegt auf der empfohlenen Flughöhe. Es soll eine günstige Route gefunden werden, die mit den Flugleistungsparametern Steigrate und Sinkrate fliegbar ist. Dazu soll berücksichtigt werden, dass bei Flughöhenänderungen, welche durch die Anpassung an die Steigrate und Sinkrate entstehen, eine andere Gefahr besteht als zuvor und dadurch eine günstigere Route gefunden werden kann.

Das zweite Augenmerk gilt der Berechnung der Route mit der geringsten Flughöhe. Zudem sollte zwischen Waffenbedrohung und Sensorbedrohung unterschieden werden. Sie müssen einzeln voneinander einstellbar sein und unabhängig in die Bewertung einfließen.

1.1.1 Fliegbarkeit durch Angabe der Steig- und Sinkrate

Trotz der Optimierungskriterien „*Bedrohung*" und „*Routenlänge*" ist es besonders wichtig, dass die Route unter der Bedingung der Fliegbarkeit gefunden wird. Ob nun aber eine Route fliegbar ist oder nicht, hängt von diversen Flugleistungsparametern ab, z.B. von der Trägheit bei der Änderung des Querneigungswinkels, der maximalen Steig- oder Sinkrate oder dem minimalen Kurvenradius. Die Berücksichtigung des minimalen Kurvenradius ist bereits integriert. In dieser Diplomarbeit werden zusätzlich noch maximale Steig- und Sinkrate berücksichtigt. Die maximale Steig- und Sinkrate geben, mit welcher Geschwindigkeit ein Luftfahrzeug maximal steigen beziehungsweise sinken kann.

1.1.2 Unterscheidung Sensorbedrohung und Waffenbedrohung

Die Unterscheidung von Bedrohungen als Sensor- oder Waffenbedrohung verbessern die Möglichkeiten der Planung. Man kann bestimmen, wie stark die einzelnen Bedrohungen in die Bewertung einfließen sollen. Sensor- und Waffenbedrohung sind zwei voneinander unabhängige Bedrohungen und können einzeln auftreten.

1.1.3 Optimierung nach der geringsten Flughöhe

Die Optimierung nach der geringsten Flughöhe bietet die Möglichkeit geländeangepasst zu fliegen. Da der Bellman-Ford-Algorithmus die Kantengewichte aufsummiert, werden kürzere Routen bevorzugt. Allerdings ist es dadurch möglich, dass über eine Geländeerhöhung geroutet wird und eine längere niedrigere Route nicht gefunden wird. Es soll nach der geringsten absoluten Flughöhe (MSL= Mean Sea Level) optimiert werden.

2. Bellman-Ford-Algorithmus

2.1 Grundbegriffe der Graphentheorie

Um weitere Betrachtungen zu unterstützen, werden hier einige Begriffe und Notationen vereinbart.

Ein Graph $G = (V, E)$, wobei $| V |$ die Anzahl der Knoten und $| E |$ die Anzahl der Kanten ist. Gewichtete Graphen sind Graphen, in denen jeder Kante ein Gewicht zugeordnet ist, wozu eine Gewichtsfunktion $w : E \to R$ verwendet wird. Für einen Graphen $G = (V, E)$ verwaltet man für jeden Knoten $v \in V$ einen Vorgänger $\pi[v]$, der entweder ein anderer Knoten oder NIL ist. Man bezeichnet $d[v]$ als Schätzung des kürzesten Pfades. Ein Vorgängerteilgraph $G\pi = (V\pi, E\pi)$ wird durch die π Werte erzeugt. Man definiert das Gewicht des kürzesten Pfades als $\delta(u, v)$.

Ein Graph $G' = (V', E')$ mit $V' \subset V$ und $E' \subset E \mid V'$ heißt Untergraph (*subgraph*) von G, wobei $E \mid V'$ genau die Kanten von E beinhaltet, die in den Knoten aus V' enden. Sind sogar die Mengen der Knoten identisch ($V' = V$), handelt es sich um einen erzeugenden Untergraphen (*spanning subgraph*).

Es gibt immer einen Knoten, in den eine gerichtete Kante mündet und einen aus dem die Kante entspringt. Die Anzahl der Kanten, die in einem Knoten v enden, ist der Eingangsgrad (*indegree*) $d_{in}(v)$ des Knotens, die Anzahl der Kanten, ausgehend von dem Knoten, ist der Ausgangsgrad (*outdegree*) $d_{out}(v)$. Knoten, deren Eingangsgrad null ist, werden auch Quelle (*source*) genannt. Wohingegen Knoten mit einem Ausgangsgrad von null als Senke (*sink*) bezeichnet werden. [1]

2.2 Kreisfreier Digraph: DAG

Ein gerichteter azyklischer Graph wird als directed acyclic graph (DAG) bezeichnet. Der Bellman-Ford-Algorithmus wird auf solchen Graphen angewendet. Es wird versprochen, dass der Bellman-Ford-Algorithmus in einer von der Kanten- und Knotenzahl linear abhängigen Zeit $\Theta(V + E)$ eine komplette Routensuche durchführt. Dies geht allerdings nur, wenn schon vor der Bewertung eines Knotens v alle seine Vorgänger bewertet sind. Wäre einer dieser Vorgänger über den Knoten v erreichbar, so würde es ihn zum direkten oder indirekten Nachfolger von Knoten v machen. Also kann dieser Knoten nicht vor Knoten v bewertet werden.

Wird jedoch akzeptiert, dass ein Knoten nach seiner Bewertung erneut bewertet werden darf, so bedeutet dies, dass jeder seiner Nachfolger erneut bewertet wird. Da die Anzahl der Zyklen von den Knoten abhinge, würde dies den angestrebten Aufwand $\Theta(V + E)$ sprengen.

Um auf Kreisfreiheit zu prüfen, bietet sich die topologische Anordnung der Knoten an, die auf jeden DAG angewandt werden kann. Dabei wird eine bijektive Abbildung erstellt, durch die jeder Knoten einen Index als Wert für seine Abhängigkeiten erhält. Wie in Abbildung 1 dargestellt ist der Index eines Knotens immer niedriger als die Indizes seiner Nachfolger, aber höher als jene seiner Vorgänger.

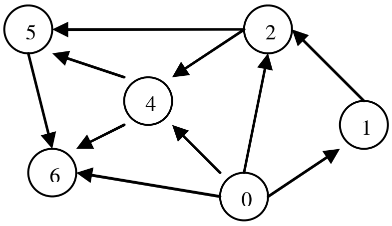


Abbildung 1 Gerichteter azyklischer Graph

Nun scheint es so, dass Knoten, die nur ausgehende Kanten haben, immer die niedrigsten und Knoten, die nur eingehende Kanten haben immer die höchsten Indizes besitzen. Diese Annahme ist nicht ganz korrekt, denn es gibt durchaus mehrere Lösungen, die sich stark voneinander unterscheiden können.

Es gibt auch solche Lösungen, bei denen einige Knoten mit vielen eingehenden Kanten einen kleineren Index besitzen als Knoten mit mehr ausgehenden Kanten. Jedoch gilt immer:

$$e(a,b) \in E \Rightarrow Index_{TopSort}(a) < Index_{TopSort}(b)$$

Die topologische Anordnung kann durch die Breitensuche entstehen. In die Tiefe geht eine solche Suche immer nur dann, wenn alle Vorgängern von einem Knoten bereits ihre Position in der topologischen Anordnung zugewiesen wurde. Erfüllen mehrere Knoten dieses Kriterium, ist es völlig irrelevant, welcher von ihnen als nächster betrachtet wird.
Jedoch kann auch Tiefensuche genutzt werden. Bei der Tiefensuche kann man die topologische Sortierung in der Zeit $\Theta(V + E)$ ausführen.

Lemma:
Ein gerichteter Graph G ist genau dann azyklisch, wenn eine Tiefensuche auf G keine Rückwärtskante liefert.

Beweis:
Angenommen, es gibt eine Rückwärtskante (u,v), dann ist der Knoten v ein Vorgänger des Knoten u im Tiefensuchwald. Es gibt daher in G einen Pfad von v nach u, und die Rückwärtskante schließt einen Zyklus.

Bei Implementierungen des Bellman-Ford-Verfahrens wird in der Regel auf ein explizites topologisches Anordnen verzichtet. [2]

2.3 Der Bellman-Ford-Algorithmus

In diesem Kapitel soll die Arbeitsweise des Bellman-Ford-Algorithmus kurz erläutert werden. Der Bellman-Ford-Algorithmus löst das Problem der kürzesten Pfade bei einem einzigen Startknoten für den allgemeinen Fall, dass die Kantengewichte negativ sein können. Jedoch finden negative Kantengewichte in dieser Diplomarbeit keinen Einsatz. Der Algorithmus beginnt mit der topologischen Sortierung des DAG. In der Regel wird jedoch bei Implementierungen des Bellman-Ford-Verfahrens auf ein explizites topologisches Anordnen verzichtet. Relevant ist es nur bei der in den Abschnitten 2.5 beschriebenen Bewertung der Knoten, wo sie stattdessen in den Bewertungsalgorithmus integriert wird. Kreisfreiheit sollte außerdem schon beim Aufspannen des Graphen gewährleistet werden. Für alle Knoten $v \in V$ verwaltet man ein Attribut $d[v]$, das eine obere Schranke für das Gewicht des kürzesten Pfads vom Startknoten s nach v ist. Man bezeichnet $d[v]$ als Schätzung des kürzesten Pfades. Man initialisiert die Schätzung der kürzesten Pfade und die Vorgänger durch die folgende Prozedur, die in Zeit $\Theta(V)$ läuft.

```
INITIALIZE-SINGLE-SOURCE $(G, s)$
    for alle Knoten $v \in V[G]$
        do $d[v] \leftarrow \infty$
           $\pi[v] \leftarrow NIL$
    $d[s] \leftarrow 0$
```

Nach der Initialisierung gilt $\pi[v] = NIL$ für alle Knoten $v \in V$ sowie $d[s] = 0$ und $d[v] = \infty$ für $v \in V - \{s\}$.

Der Prozess des Relaxierens einer Kante (u,v) besteht darin, zu testen, ob wir den bisher gefundenen kürzesten Pfad nach v verbessern können, indem wir durch u gehen, und wenn dies der Fall ist, die Attribute $d[v]$ und $\pi[v]$ entsprechend zu aktualisieren. Ein Relaxationsschritt kann den Wert der Schätzung des kürzesten Pfades verringern und das Vorgängerattribut $\pi[v]$ des Knotens v aktualisieren. Der folgende Code führt einen Relaxationsschritt auf der Kante (u,v) aus.

```
RELAX $(u,v,w)$
        if $d[v] > d[u] + w(u,v)$
                then $d[v] \leftarrow d[u] + w(u,v)$
                     $\pi[v] \leftarrow u$
```

Der Bellman-Ford-Algorithmus ruft INITIALIZE-SINGLE-SOURCE auf und relaxiert dann die Kanten sukzessive. Beim Bellman-Ford-Algorithmus wird jede Kante mehrmals relaxiert. Die Relaxation ist die einzige Methode, durch die die Schätzung der kürzesten Wege und die Vorgänger geändert werden können.

```
DAG-SHORTEST-PATH $(G,w,s)$
Führe eine topologische Sortierung der Knoten von $G$ durch
INITIALIZE-SINGLE-SOURCE $(G,s)$
for alle Knoten $u$ in topologisch sortierter Reihenfolge
        do for alle Knoten $v \in adj[u]$
                do RELAX $(u,v,w)$
```

Die folgenden Graphiken verdeutlichen die Arbeitsweise des Bellman-Ford-Algorithmus auf einem gerichteten azyklischen Graphen (DAG). Der Startknoten ist s und der Zielknoten ist z . In den Knoten sind die laufenden Kosten (d-Werte) eingetragen.

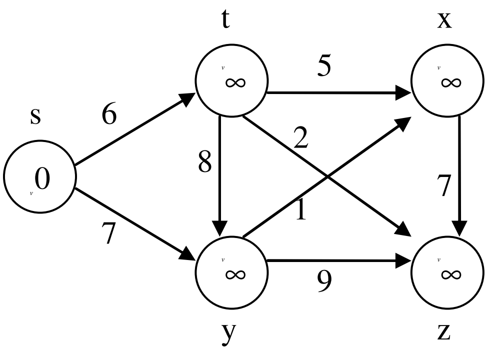


Abbildung 2 Bsp. Bellman-Algorithmus Ausgangssituation

Erster Schritt: Alle Nachfolgerknoten des Startknoten s werden betrachtet und bewertet.

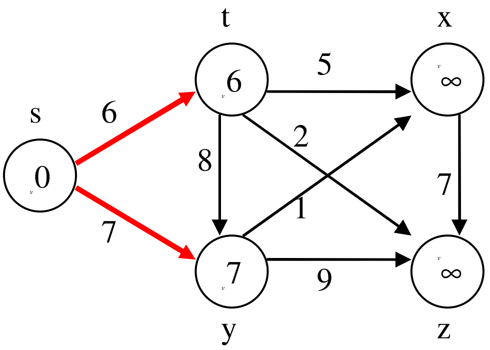


Abbildung 3 Bsp. Bellman-Algorithmus erster Schritt

Zweiter Schritt: Alle Nachfolgerknoten des Knoten t werden betrachtet und bewertet.

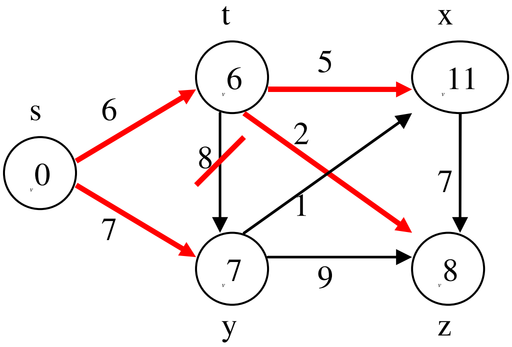


Abbildung 4 Bsp. Bellman-Algorithmus zweiter Schritt

Da der Knoten y günstiger über die Kante(s, y) als über die Kante(t, y) zu erreichen ist, wird die Kante(t, y) nicht weiter betrachtet. Im dritten Schritt werden alle Nachfolgeknoten des Knotens y betrachtet und bewertet. Da der Knoten z über die Kante(t, z) günstiger zu erreichen ist, wird die Kante(y, z) gestrichen. Dasselbe geschieht mit der Kante(t, x), da der Knoten x über die Kante(y, x) günstiger zu erreichen ist.

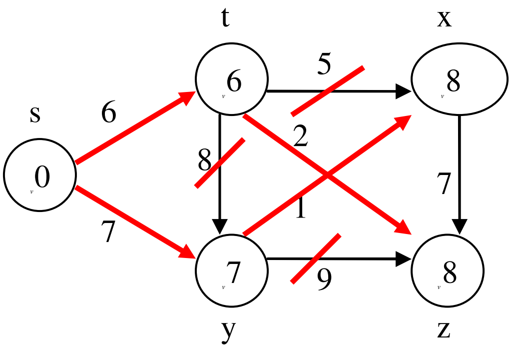


Abbildung 5 Bsp. Bellman-Algorithmus dritter Schritt

Im vierten Schritt werden die Nachfolger des Knoten x betrachtet und bewertet.

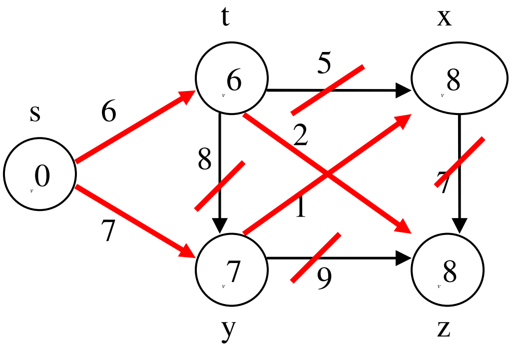


Abbildung 6 Bellman-Algorithmus vierter Schritt

Da der Zielknoten durch die Kante(t, z) günstiger zu erreichen ist als über die Kante(x, z) wird diese nicht beachtet. Die kürzeste Route von Startknoten s zu Zielknoten z sieht wie folgt aus.

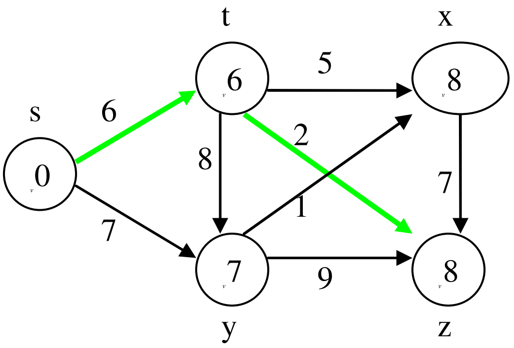


Abbildung 7 Bellman-Algorithmus Lösung günstigste Route

Das folgende Theorem zeigt, dass die Prozedur DAG-SHORTEST-PATHS die kürzesten Pfade korrekt berechnet.

Theorem:

Falls ein gerichteter Graph $G = (V, E)$ mit dem Startknoten s keine Zyklen enthält, dann gilt bei Terminierung der Prozedur DAG-SHORTEST-PATHS $d[v] = \delta(s, v)$ für alle Knoten $v \in V$ und der Vorgängerteilgraph $G\pi$ ist ein Baum kürzester Pfade.

Beweis:

Man zeigt zunächst, dass für alle Knoten $v \in V$ bei der Terminierung $d[v] = \delta(s, v)$ gilt. Wenn v von s aus nicht erreichbar ist, dann gilt wegen der Kein-Pfadeigenschaft $d[v] = \delta(s, v) = \infty$. Nehmen wir nun an, dass v von s aus erreichbar ist, sodass es einen kürzesten Pfad $p = \langle v0, v1, ..., vk \rangle$ mit $v0 = s$ und $vk = v$ gibt. Da wir die Knoten in topologischer Reihenfolge verarbeiten, sind die Kanten auf p in der Reihenfolge $(v0, v1), (v1, v2), ..., (vk - 1, vk)$ sortiert. Aus der Pfadrelaxationseigenschaft folgt, dass für $i = 0,1,...,k$ bei Terminierung $d[v] = \delta(s, vi)$ gilt. Schließlich folgt aus der Vorgängerteilgraph-Eigenschaft, dass $G\pi$ ein Baum kürzester Pfade ist. Die Laufzeit ist: [2]

$$\Theta(V + E)$$

2.4 Bewertung der Kanten

Eine gerichtete Kante beschreibt den Weg von einem Knoten zu einem seiner Nachfolger. Über die Wegstrecke fallen Kosten an, welche als Gewichte (weights) bezeichnet werden. Diese Gewichte können beim Bellman Algorithmus auch negativ ausfallen, was aber in dieser Diplomarbeit keine Anwendung findet.

Es gibt verschiedene Verfahren um festzulegen, von welchem Knoten zu welchem Nachbarn eine Kante entstehen soll und in welche Richtung sie zeigt. Da dies nicht der Hauptbestandteil meiner Diplomarbeit ist beschränke ich mich darauf, nur das eingesetzte Verfahren zu beschreiben. [3]

2.4.1 Aufspannen des Graphen

Für die Bewertung wird ein Graph aufgespannt. Als Basis für das Aufspannen des Graphen wird eine Gitterstruktur verwendet, die mit der Größe der ausgewählten Landschaft übereinstimmt. In diesem Gitter wird auf jedem Gitterpunkt genau ein Knoten angenommen. Die Auflösung des Gitters ist über die GUI individuell einstellbar. Einem Knoten stehen potenziell sechzehn Nachbarn zu Verfügung, wobei acht direkte Nachbarn und acht entfernte Nachbarn sind (siehe Abbildung 8. rechts). Die entfernten Nachbarn werden über einen Rösselsprung erreicht. Der Rösselsprung ist ein Bewegungsablauf der dem im Schach vorgeschriebenen Zug des Springers entspricht. Man geht zwei Gitterpunkte einer Linie oder Reihe in eine Richtung und einen Gitterpunkt zur Seite (siehe Abbildung 8. links).

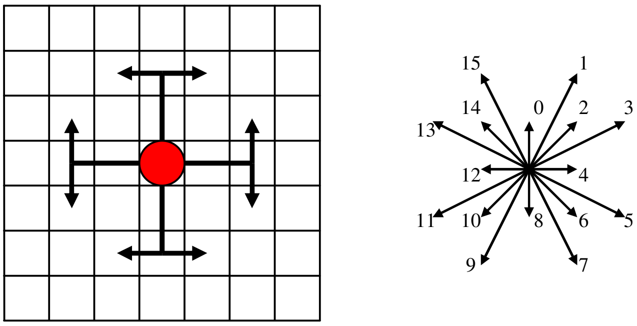
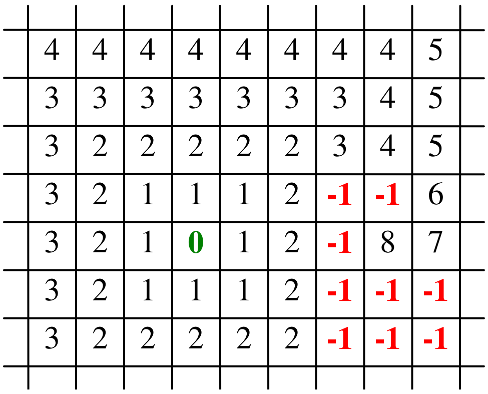


Abbildung 8 Rösselsprung und Nachbarknoten

Wird jeder Knoten nun mit all seinen Nachbarn durch gerichtete Kanten in alle Richtungen verbunden, entstehen zwangsläufig gerichtete Kreise, die allerdings für den Bellman-Ford-Algorithmus hinderlich sind. Es wird ein DAG benötigt, der trotz seiner Einschränkung der Kreisfreiheit ein Höchstmaß an optimalen Routen zu den einzelnen Knoten liefert.

Als wünschenswert gilt hier ein System, bei dem alle Kanten von einem bestimmten Knoten weg führen. Diese Quelle entspricht dem gewünschten Endpunkt der Route. Um sicherzustellen, dass jede Kante vom Austrittspunkt wegführt, wird über Breitensuche (*breadth first search*) jedem Knoten eine Priorität zugewiesen. Der Austrittspunkt beginnt bei null, und jeder seiner acht direkten Nachbarn erhält den Wert eins. Jeweils um eins höher als die geringste Priorität in seiner direkten Nachbarschaft. Knoten, die in Sperrgebieten liegen, oder solche, die nicht erreichbar sind, verbleiben auf ihrem Initialwert -1.



4	4	4	4	4	4	4	4	5
3	3	3	3	3	3	3	4	5
3	2	2	2	2	2	3	4	5
3	2	1	1	1	2	**-1**	**-1**	6
3	2	1	**0**	1	2	**-1**	8	7
3	2	1	1	1	2	**-1**	**-1**	**-1**
3	2	2	2	2	2	**-1**	**-1**	**-1**

Abbildung 9 Priorisierung beim Aufspannen des Graphen

Bei Abbildung 9 handelt es sich um eine beispielhafte Darstellung der Priorisierung. Die Quelle ist grün und Sperrgebiete sind rot dargestellt. Zu erkennen ist deutlich, dass sich die Prioritäten von Knoten in direkter Nachbarschaft maximal um eins unterscheiden.

Nach der Priorisierung werden die Kanten erstellt. Kanten werden immer von Knoten niederer Priorität zu solchen höherer Priorität gezogen.

Und auch bei gleicher Priorität wird eine Kante erstellt, allerdings muss der Endknoten der Kante weiter von der Quelle entfernt sein als der Anfangspunkt der Kante, da sonst wieder gerichtete Zyklen entstehen würden (siehe Abbildung 10).

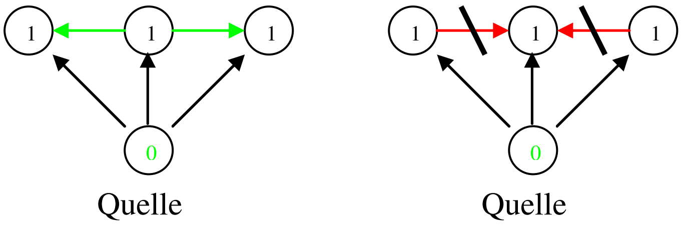


Abbildung 10 Kanten zwischen Knoten mit gleicher Priorität

Zu beachten sind zudem Knoten, die mit dem Rösselsprung erreicht werden. Sie überspannen mit ihrer Kante zwei direkte Nachbarknoten, so genannte Transitpunkte (Abbildung 11). Bei der späteren Gewichtung der Kanten fließt ein geringer Anteil der Gewichte der überspannten Knoten mit ein.

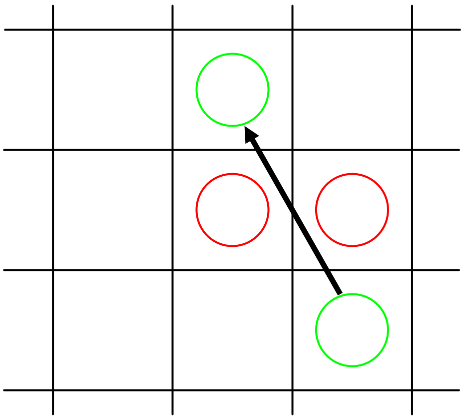


Abbildung 11 Transitpunkte

Grün markiert sind Anfangs- und Endpunkt. Der Endpunkt wird über einen Rösselsprung erreicht. Rot markiert sind die Transitpunkte. Die Kante, welche durch den Rösselsprung entsteht, geht durch Bereiche der Transitpunkte. Deshalb fließt ein kleiner Anteil ihrer Gewichtung bei der Gewichtung der Kante mit ein.

Jede Kante wird in der sogenannten Adjazenzliste gespeichert, eine Liste, in der für jeden Knoten die ausgehenden Kanten und ihre Endknoten festgehalten sind. In keinem Fall darf eine Kante einen Knoten überspannen, der zu einem Sperrgebiet gehört.

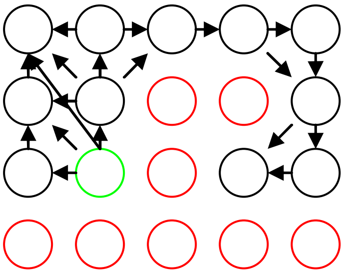


Abbildung 12 Gerichtete Kanten

Abbildung 12 zeigt, dass gerichtete Kanten so weit wie möglich zu den Nachbarn gezogen werden. Rot markierte Kreise sind Sperrgebiete. Keine Kante darf ein Sperrgebiet überspannen.

Durch die Priorisierung der Knoten und den Vergleich der Entfernungen bei gleicher Priorität ist sichergestellt, dass keine gerichteten Kreise entstehen können, da es keinen einfachen gerichteten Weg geben kann, welcher von einem Endpunkt einer Kante zu ihrem Anfangspunkt zurück führt.

Das Verfahren terminiert, nachdem jeder Knoten genau einmal expandiert und damit sämtliche Kanten $e \in E$ erstellt wurden. Expandieren bedeutet, dass alle ausgehenden Kanten $e_i = (a, b_i)$ für $i \in [0, d_{out}(a)[$ des Knotens a betrachtet werden. Der Aufwand zum Aufspannen des Graphen hängt linear von der Anzahl der Kanten und Knoten ab. [3]

2.4.2 Kosten einer Kante

Da die Gewichte einer Kante nicht von der Struktur des Graphen abhängen, sondern allein von den Gewichten der Anfangsknoten, Endknoten und dem zurückgelegten Weg, können die Kanten nacheinander ohne spezielle Anordnung gewichtet werden. [3]

Die angewandte Kostenfunktion hängt stark von der Problemstellung ab. Es kommt darauf an, ob nach der geringsten Flughöhe, Sichtbarkeit, Länge usw. optimiert werden soll. Ist der Aufwand für die Berechnung eines Gewichtes für eine Kante konstant, ist das Bewertungssystem linear von der Anzahl der Kanten abhängig. Das Aufspannen des Graphen, welches mit Breitensuche geschieht, ist abhängig von der Anzahl der Knoten, sowie von den Kanten. Somit ergibt sich eine Komplexität von:

$$O(|V| + |E|)$$

Im vorliegenden Fall ist die Anzahl der Kanten auf sechzehn beschränkt und somit die Anzahl der Kanten linear abhängig von der Anzahl der Knoten. Deshalb ist die Komplexität nur noch:

$$O(|V|)$$

2.5 Bewertung der Knoten

Mittels einer kompletten Berechnung der Abstände aller Knoten des Graphen wird sichergestellt, dass zu jeder Zeit der Abstand zum Anfangspunkt bekannt ist. Die Bewertung sieht wie folgt aus.

Am Anfang werden die Eingangsgrade (*indegrees*) $d_{in}(v)$ der einzelnen Knoten benötigt. Eine Liste mit den Eingangsgraden kann schon beim Aufspannen des Graphen erzeugt werden, aber auch danach kann man die Eingangsgrade schnell ermitteln. Mit linearem Aufwand würde man einfach über die Adjazenzliste iterieren und einen Zähler für jeden Endknoten inkrementieren.

Während des laufenden Bewertungsverfahrens wird die Liste der Eingangsgrade dazu verwendet, die Anzahl der noch nicht betrachteten eingehenden Kanten zu sichern und nicht die der tatsächlich eingehenden Kanten.

In der Kostenmatrix werden für jeden Knoten die minimalen laufenden Kosten gespeichert. Zudem gibt es noch eine Liste, die alle direkten Vorgängerknoten zu jedem Knoten speichert.

Die Kostenmatrix wird mit dem Maximalwert initialisiert, damit die erste gefundene Route zu dem Knoten auf jeden Fall günstiger ist als der in der Kostenmatrix eingetragene Wert. Eine Ausnahme bildet hier der Anfangspunkt des Graphen, dessen Kosten gleich null sind.

Die Liste, in der die Vorgängerknoten gespeichert werden, wird mit einem Wert initialisiert, den kein existierender Knoten hat. Dafür bietet sich die -1 an. Wiederum bietet der Anfangsknoten eine Ausnahme, denn er wird mit seinem eigenen Index initialisiert. Somit ist der Anfangsknoten der einzige Knoten, dessen eigener Index dem seines Vorgängers entspricht.

Durch die Festlegung genau eines Vorgängers für einen Knoten, der aber mehrere Nachfolger haben kann, entsteht ein Untergraph in Form eines Baumes (*tree*), dessen Wurzel (*root*) der gewählte Anfangsknoten des Graphen ist.

Eine Kante die nicht als Vorgänger ihres Endknotens festgelegt wurde, wird nach der vollständigen Bewertung der Knoten vernachlässigt, da sie kein Teil der optimalen Route sein kann.

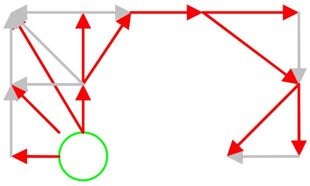


Abbildung 13 Baumstruktur des erzeugenden Untergraphen

In Abbildung 13 ist ein solcher Untergraph, der dabei entstehen kann, abgebildet. Rot sind die Kanten dargestellt, die für die optimalen Routen relevant sind. Grau wiederum sind die Kanten abgebildet, welche in keiner optimalen Route enthalten sind und nach der vollständigen Bewertung des Graphen vernachlässigt werden. Durch das Ausblenden der grauen Kanten zeigt sich die Baumstruktur, die jeden von der Quelle erreichbaren Knoten mit der Quelle verbindet. Die Verbindungen sind optimale Routen durch den Graph.

Zur Bewertung der Knoten selbst wird eine Schlange[1] verwendet. In dieser Schlange sind alle Knoten enthalten, deren verbleibender Eingangsgrad (also die Anzahl eingehender Kanten von Knoten, die noch nicht expandiert wurden) gleich null ist. Die laufenden Kosten der Knoten in der Schlange sind bewertet und minimal. Knoten, deren Nachfolger bewertet werden, sind zur Expansion bereit.

Sind nun alle Datenstrukturen angelegt und initialisiert, beginnt die eigentliche Bewertung. Solange sich noch Knoten in der Schlange befinden, wird ein Knoten a aus der Schlange entnommen und expandiert. Für jede dieser Ausgangskanten wird das laufende Gewicht des Knoten a auf das Gewicht der Kante e_i addiert. Sollten die neu errechneten laufenden Kosten kleiner sein, als die bisher für Knoten b_i gespeicherten, werden die bisherigen Kosten durch die neuen Kosten ersetzt. Dadurch wird Knoten a zum neuen Vorgänger von Knoten b_i.

---

[1] Es kann auch ein Stapel genutzt werden. Es muss eine Datenstruktur variabler, aber bekannter maximaler Größe verwendet werden, die schnelles Einfügen und Entfernen der Elemente erlaubt. Die Reihenfolge spielt dabei keine Rolle, da alle Knoten gleichwertig sind.

Sollten die errechneten Kosten jedoch nicht kleiner sein als die bisher für Knoten b_i gespeicherten, bleiben die Einträge unverändert.

In beiden Fällen wird der Wert der Eingangskanten (indegrees) $d_{in}(v)$ dekrementiert, da sich die Anzahl der noch nicht betrachteten Kanten um eins reduziert hat. Wird der Wert null erreicht, so wird auch dieser Knoten in die Schlange eingereiht. Nun kann dieser Knoten problemlos expandiert werden, da es keinen weiteren Vorgänger gibt und dadurch garantiert ist, dass die laufenden Kosten minimal sind, denn es kann kein Vorgänger mit geringeren laufenden Kosten gefunden werden. [3]

Am Ende war jeder von der Quelle erreichbare Knoten des Graphen einmal in der Schlange. Die Komplexität dieses Verfahrens ist linear abhängig von der Knoten- und der Kantenanzahl im Graphen.

$$O(|V| + |E|)$$

Im vorliegenden Fall ist die Anzahl der Kanten auf sechzehn beschränkt und somit die Anzahl der Kanten linear abhängig von der Anzahl der Knoten. Deshalb ist die Komplexität nur noch:

$$O(|V|)$$

2.6 Wegsuche im Graph

Nach der Knotenbewertung ist nun bekannt, welche Kosten auf dem Weg von jedem Knoten im Graph zur Quelle anfallen werden. Ausgehend von jedem Knoten ist der Vorgänger bekannt, welcher als nächstes passiert werden muss, damit die Quelle des Graphen am günstigsten erreicht wird. Eine Iteration über die Routenpunkte zum jeweiligen Vorgänger ist ausreichend, um die optimale Route zusammenzufassen. Im Worst Case sind alle Knoten im Graph an der abschließenden Suche beteiligt, so dass die Komplexität linear von der Knotenanzahl abhängt. [3]

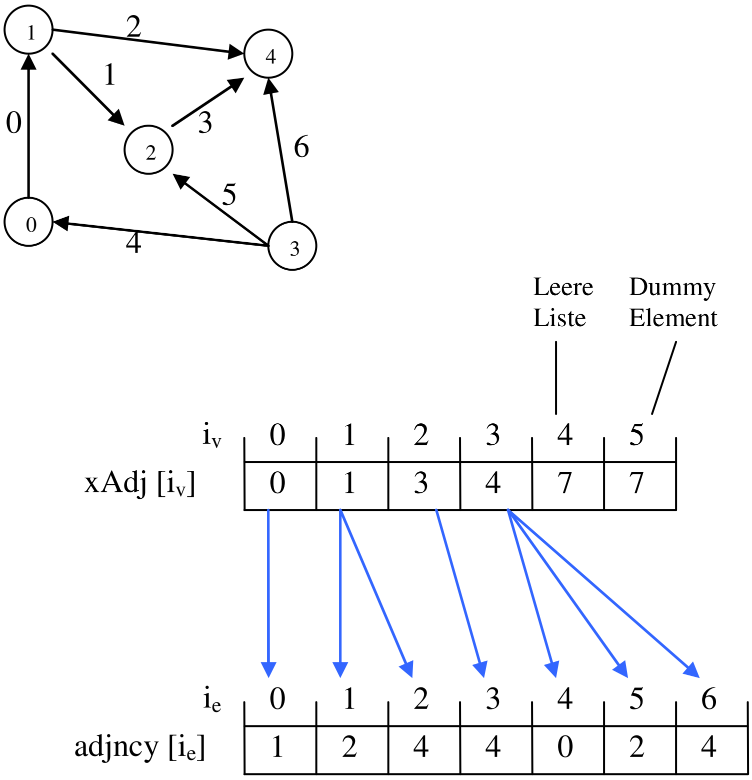
$$O(|V|)$$

3. Datenstruktur Forward Star

Die wohl beste Methode einen Digraph zu implementieren ist die Forward Star Datenstruktur. Sie besteht aus zwei eindimensionalen Matrizen, welche die Adjazenzliste (Nachfolgerliste) beschreiben, also welcher Knoten mit welchem Adjazent (Nachfolger) verbunden ist. Unabhängig von der Position oder Struktur des Graphen wird jedem Knoten ein Index $i_v \in [0, n[$ zugewiesen.

Die erste Matrix (*xAdj*) besitzt für jeden Knoten einen Eintrag und zusätzlich einen Dummyeintrag, der die Liste terminiert, sprich $n+1$ Einträge. Jedes dieser Elemente verweist auf die Stelle in der zweiten Matrix (*adjncy*), an der die Endknoten der ausgehenden Kanten beginnen. Die zweite Matrix (*adjncy*) besitzt für jede Kante genau ein Element. In diesen Elementen werden die Endknoten der Kanten gespeichert. Es wird für jede Kante ein eindeutiger Index $i_e \in [0, m[$ vergeben, der die Daten der Kanten referenziert.

Die Abbildung 14 verdeutlicht die Struktur anhand eines Beispielgraphen. Die Symbole i_v und i_e wurden gewählt, um deutlich zu machen, dass es sich um Abhängigkeiten von den Indizes der Knoten und der Kanten handelt.



i_v	0	1	2	3	4	5
xAdj [i_v]	0	1	3	4	7	7

i_e	0	1	2	3	4	5	6
adjncy [i_e]	1	2	4	4	0	2	4

Abbildung 14 Datenstruktur Forward Star

Zu erkennen ist, dass die Differenz des Elements eines Knotens v in *xAdj* zu dem nachfolgenden Element mit der Anzahl der ausgehenden Kanten von v, also $d_{out}(v)$ übereinstimmt. Außerdem besitzt mindestens das letzte Element der Matrix *xAdj* (das Dummy-Element) einen Wert, der m, der Anzahl der Kanten, entspricht.

Der Vorteil der Forward Star Datenstruktur liegt darin, dass man nur zwei eindimensionale Felder braucht um eine komplette Adjazenzliste zu modellieren. [3]

4. Visibility

Visibility oder Sichtbarkeit ist die visuelle Erkennbarkeit eines Flugkörpers. Es wird im Grunde geschaut, ob in unserem Fall das Luftfahrzeug in der Sonne fliegt und dadurch visuell für das menschliche Auge leichter sichtbar ist, oder ob das Luftfahrzeug im Schatten eines Berges oder Hügels fliegt und dadurch für das menschliche Auge schwerer sichtbar ist.

Die Sonne kann als Punkt-Lichtquelle mit unendlichem Abstand betrachtet werden und folglich können alle Strahlen, die zu einem Rasterfeld gelangen, als parallel laufend betrachtet werden. Zur Erleichterung der Berechnung wird eine „Belichtungsebene“ SP senkrecht zu den Sonnenstrahlen berücksichtigt. Alle Sonnenstrahlen durchlaufen diese Ebene im rechten Winkel. Beim Überprüfen der Projektion einer Gitterzelle über diese Ebene in Richtung Sonne folgend, kann man entscheiden, ob ein Punkt in der Sonne oder im Schatten eines anderen Punktes liegt.

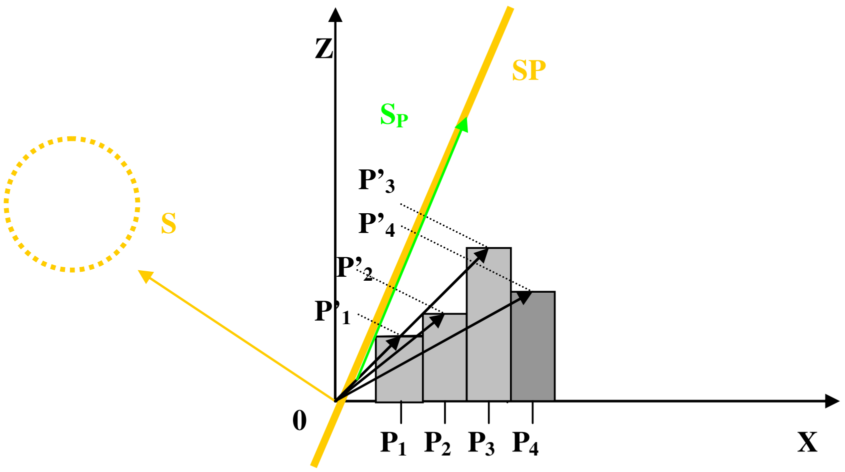


Abbildung 15 Visibility

Da die Sonne als Punktquelle in unendlichen Abstand betrachtet werden kann, sind alle Sonnenstrahlen parallel und durchlaufen die Ebene SP senkrecht. Indem wir die Projektion eines Rasterfelds über diese Ebene prüfen, können wir feststellen, ob ein Punkt in der Sonne oder im Schatten einer vorhergehenden Zelle liegt.

In Abbildung 15 ist die Projektion von P_1, P'_1, höher als alle Vorgänger (da es der erste betrachtete Punkt ist), und ist somit in der Sonne. Das gleiche gilt für Punkt P_2 und P_3, allerdings hat P_4 eine niedrigere Projektion als P_3 und liegt somit im Schatten von P'_3. Die Projektion eines Punktes P_i auf der Belichtungsebene SP ist ein Punktprodukt des Vektors $\overrightarrow{OP_i}$ und des Vektors $\overrightarrow{Sp}$, welcher ein Einheitsvektor senkrecht zu Vector $\vec{S}$ ist, der Einheitsvektor der Sonne. [4]

4.1 Einheitsvektor senkrecht zu den Sonnenstrahlen

Zusätzlich führt man den Winkel θ ein, welcher die Lage der Belichtungsebene angibt und beschreibt gewissermaßen, aus welcher Richtung die Strahlen kommen.

Man unterscheidet zwei Fälle:

- $\theta > 0$

$\overrightarrow{Sp^W} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$ Wenn die Berechnung von der West- in die X-Richtung erfolgt ist.

$\overrightarrow{Sp^N} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$ Wenn die Berechnung von der Nord- in die Y-Richtung erfolgt ist.

- $\theta < 0$

$\overrightarrow{Sp^E} = \begin{bmatrix} \cos\theta \\ -\sin\theta \end{bmatrix}$ Wenn die Berechnung für die Ost- in die X-Richtung erfolgt ist.

$\overrightarrow{Sp^S} = \begin{bmatrix} \cos\theta \\ -\sin\theta \end{bmatrix}$ Wenn die Berechnung für die Süd- in die Y-Richtung erfolgt ist.

Die folgenden Abbildungen zeigen den Winkel θ in X-Richtung und Y-Richtung.

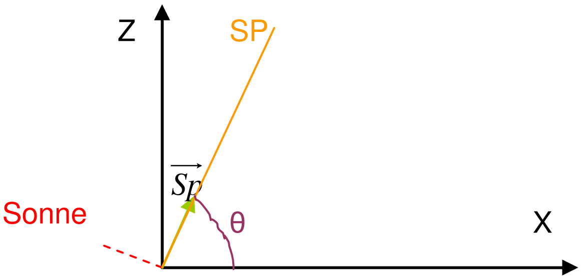


Abbildung 16 Winkel in X-Richtung

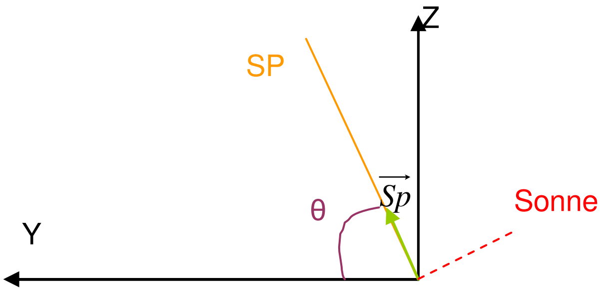


Abbildung 17 Winkel in Y-Richtung

Die Abbildung 18 zeigt die Belichtung für einen positiven Winkel θ und einen negativen Winkel θ (beide haben den gleichen Absolutwert). Es muss davon ausgegangen werden, dass die Szene von Westen und Osten, aber auch von Süden und Norden bestrahlt werden kann. [4]

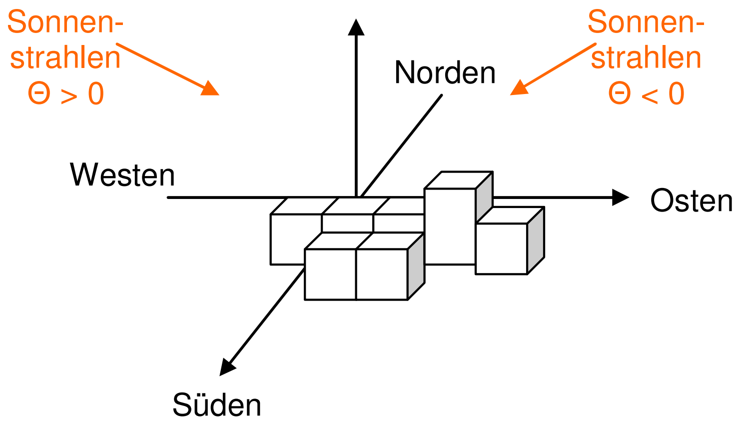


Abbildung 18 Belichtung positiv und negativ

4.2 Umstrukturierung des Visibility-Algorithmus

Der bereits existierende Visibility-Algorithmus berechnete die Visibility komplett für jeden Punkt und gab diese dann in der sogenannten SkyView Matrix zurück. Diese Matrix enthielt für jeden Punkt einen Visibility-Wert.

Für diese Diplomarbeit musste man den Algorithmus so umstellen, dass er für einen bestimmten Punkt mit einer bestimmten Höhe die Visibility berechnet. Da man für jeden Punkt mehrere Höhen haben könnte (siehe Abschnitt 7.3.4) muss man für jeden Punkt mehrmals die Visibility berechnen. Da hilft es nicht, dass der bisherige Algorithmus die Visibility-Werte in einer Matrix speichert, da sie mit einer tatsächlichen Höhe berechnet wurden. Anschließend speichert man den Visibility-Wert an der dazugehörigen Kante. Somit berechnet man die Visibility nicht mehr für jeden Punkt, sondern für jede Kante.

Code:

Zuvor:

```
CalcVisibility(Matrix)
    for jeden Punkt
        berechne Visibility
        Matrix[Punkt]=Visibility
    return Matrix
```

Umstrukturiert:

```
CalcVisibility(Punkt, Höhe )
    berechne Visibility
    return Visibility
```

5. Sensorbedrohung und Waffenbedrohung

Im Rahmen der Bedrohungsanalyse wird als Sensor ein Gerät definiert mit welchem Flugkörper erkannt und ihre Position bestimmt werden können. Normalerweise sind Radargeräte solche Sensoren.

Ein Sensor ist ein Element in einem Bedrohungsszenario. In dieser Arbeit werden nur Sensoren und Waffenstellungen betrachtet, die wenigstens für eine gewisse Zeit fest installiert sind. Die Reichweite wird durch den Radius bestimmt. Das Bedrohungsgebiet ist somit eine Kugel um den Sensor herum. In dieser Diplomarbeit wird die Bedrohungsintensität durch Werte zwischen 0 und 255 angegeben. Je niedriger der Wert, desto ungefährlicher ist der Sensor.

Eine Waffe ist ebenso ein Element des Bedrohungsszenarios. Die Reichweite wird durch den Radius bestimmt. Das Bedrohungsgebiet ist somit eine Kugel um den Sensor herum. In dieser Diplomarbeit wird die Bedrohungsintensität durch Werte zwischen 0 und 255 angegeben. Je niedriger der Wert, desto ungefährlicher ist die Waffe.

Da die meisten Sensoren und Waffenstellungen auf der Erdoberfläche installiert sind, ist der Bedrohungsraum um sie herum tatsächlich eher eine Halbkugel. Durch Geländeerhebungen und Geländeeinschnitte ergeben sich weitere Beschränkungen des Bedrohungsraums.

Die Berechnung des Sensor/Waffenschattens erfolgt, indem man einen Bereich um den gegebenen Sensor mit einer Line-of-sight Berechnung ablichtet. Dieser Bereich besteht aus einer Anzahl von quadratischen Rahmen (siehe Abbildung 19). Die Zahl dieser Rahmen hängt vom Radius des Sensors und der Anzahl der Gitterpunkte pro Meter ab. Um einen geschlossenen Rahmen zu bilden, muss die Anzahl der Punkte ungerade sein. Da die Rahmen an die Gitterstruktur geordnet werden, liegt ihre Mitte (der Sensor) genau in einem Gitterpunkt.

```
+------------+
| +--------+ |
| | +---+ | |
| | | S | | |
| | +---+ | |
| +--------+ |
+------------+
```

Abbildung 19 Unterscheidung Sensor und Waffe geschlossener Rahmen

Die Berechnung der Anzahl der Rahmen hängt vom Radius und von der Anzahl der Gitterpunkte ab. Um sicher zu gehen, dass man alle vom Sensor betroffenen Punkte berücksichtigt hat, wird zur nächst größeren Zahl aufgerundet.

```
//Berechnung Anzahl der Rahmen
MRR::int32 frames = MATHfunction::Ceil(maxGridPointsPerMeter_ * radius);
```

Es wird jeder Punkt auf jedem Rahmen betrachtet und bewertet.

```
MRR::int32 frame = 1;
 while (frame <= frames)
 {
  //Von Oben nach Unten
  for (MRR::int32 scanOffsetY = -frame; scanOffsetY <= frame; scanOffsetY++)
  {
    // Von Links nach Rechts
    MRR::int32 scanOffsetX = -frame;
    while (scanOffsetX <= frame)
    {
     //Wenn der Punkt im Rahmen liegt
     if ((scanOffsetY == -frame) || (scanOffsetY == frame)
      || (scanOffsetX == -frame) ||(scanOffsetX == frame))
     {
         //Betrachten und bewerten des Punktes
         //scanOffsetX inkrementieren
          scanOffsetX++;
     }
   }
}
```

Die Bewertung der Punkte ist unterschiedlich, da Punkte die weiter vom Mittelpunkt und somit auch weiter vom Sensor oder der Waffe entfernt sind, weniger gefährdet sind, als die Punkte, welche nahe am Mittelpunkt und somit sehr nahe am Sensor oder an der Waffe liegen. Die Bedrohung nimmt nach außen hin ab, da die Gefahr eines Abschusses oder einer genauen Positionierung abnimmt. Der Mittelpunkt bekommt das Maximum der Bewertung zugewiesen, da dieser direkt über dem Sensor oder der Waffe liegt und somit der Flugkörper sehr leicht zu treffen und erkennen ist.

```
// Berechnung des Gefahrenverlusts
MRR::int32 engagementThreat = MATHfunction::Round(engagement.GetIntensity()
* (radius / (radius + distance)));

// Addition von Gefahren von sich  Überlagernden Sensoren (Maximum ist 255)
engagementThreat_[scanIndex] = __min(254, engagementThreat_[scanIndex] +
engagementThreat);
```

Außerdem ist noch maßgeblich, ob ein Sensor oder eine Waffe durch eine Geländeerhöhung abgeschirmt ist und dadurch für den Flugkörper ein ungefährlicher Bereich entsteht. Die abgeschirmten Bereiche haben keine Sensor- oder Waffenbedrohung. Aus Bequemlichkeit wird bei der Line-of-sight Berechnung als Raum der Bedrohung von einem Zylinder ausgegangen und nicht von einer Kugel. Zylinder und Kugel können als äquivalent betrachtet werden, weil die in dieser Arbeit betrachteten Bedrohungsszenarien eine relativ geringe Flughöhe des Luftfahrzeugs über Grund haben. (siehe Abbildung 20)

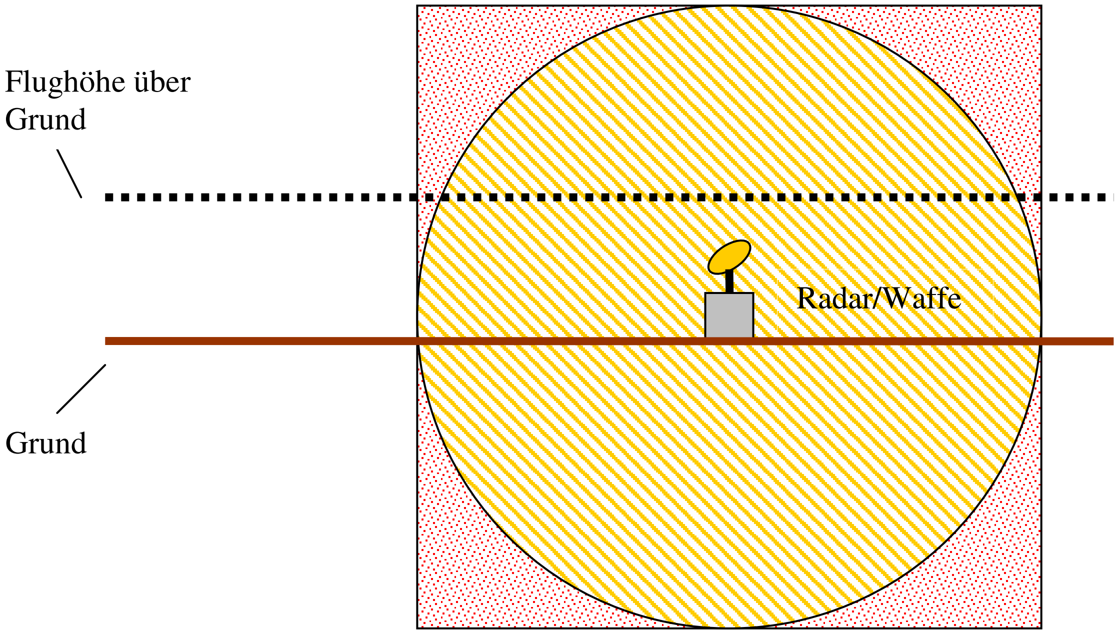


Abbildung 20 Bedrohung als Zylinder

Die Berechnung des Radarschattens, wird durch Winkelberechnung realisiert. Es wird der Winkel zwischen Radargerät und dem aktuellen Punkt berechnet (angleOfGround). Danach wird verglichen, ob der Winkel des aktuellen Punktes (angleOfGround) größer ist als der seines Vorgänger-Punktes (angleOfSight). Ist der Winkel des aktuellen Punktes (angleOfGround) größer als sein Vorgänger so wird dieser Wert in die Angle Matrix gespeichert. Ist der Winkel des Aktuellenpunktes (angleOfGround) kleiner als der Winkel seines Vorgängers (angleOfSight) so wird der Wert des Vorgängers in der Angle Matrix gespeichert und die maximale Höhe, die man fliegen kann ohne gesehen zu werden, wird neu gesetzt.

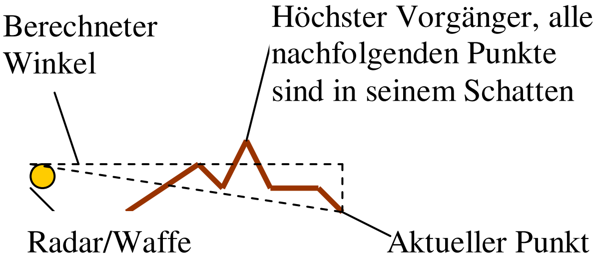


Abbildung 21 Berechnung des Radarschattens

```
//Beschaffen der Höhe
MRR::int16 height = elevationMap.GetElevation(scanIndex);

//Berechnung des Winkels, des aktuellen Punktes
MRR::float64 angleOfGround = MATHfunction::ArcTanD2(height - sensorHeight,
distance);

// Man geht von 2 Vorgängern aus die in der Sicht liegen
//könnten. Von ihnen wird der kleiner genommen, da er schlechter ist.
MRR::float64 angleOfSight = __min(angle[predAngleIndex1],
angle[predAngleIndex2]);

// Vergleich der Winkel
if (angleOfGround > angleOfSight)
{
      //Wenn der aktuelle winkel größer ist, ist er neues maximum
      angle[scanAngleIndex] = angleOfGround;
}

else
{
      //Wenn Winkel kleiner als Vorgänger Winkel dann ist der
      //Winkel des Vorgängers das Maximum
      angle[scanAngleIndex] = angleOfSight;

      //Setzen der maximalen Höhe in der man nicht gesehen wird.
      height = MATHfunction::Round(sensorHeight +
      MATHfunction::TanD(angleOfSight) * distance);
}
```

Durch das Setzen der Höhe, die maximal geflogen werden kann, ohne gesehen zu werden, wird später entschieden, ob dieser Punkt ein Schatten ist oder nicht. Man vergleicht hierzu die berechnete Flughöhe mit der maximalen Höhe in der man nicht gesehen wird. Sollte die Flughöhe geringer oder gleich sein, ist der Punkt ungefährdet. Das folgende Codebeispiel zeigt den Vergleich für jeden Punkt und somit die Feststellung, ob ein Punkt im Bedrohungsschatten liegt oder nicht.

```
//Für jeden Punkt
for (i = 0; i < dimX_*dimY_; i++)
 {
      //Beschaffen der maximalen Höhe, in der man nicht gesehen wird, für
      //Sensoren.
    MRR::float32 sensorAreaHeightMSL = sensorMap.GetThreatHeightMSL(i);

      //Beschaffen der maximalen Höhe, in der man nicht gesehen wird, für
      //Waffen.
    MRR::float32 engagementAreaHeightMSL =
    engagementMap.GetThreatHeightMSL(i);

      //Beschaffen der Flughöhe.
    MRR::float32 fleightHeightMSL = flightHeight_ +
    zeroPlane.GetZeroHeight(i);

      //Für Sensor
      //Vergleich ob die Flughöhe höher ist als die maximale Höhe, in der
      //man nicht gesehen wird, für Sensoren.
    if (fleightHeightMSL >= sensorAreaHeightMSL)
    {
      //Wenn die Flughöhe größer oder gleich der maximalen Höhe, in der
      //man nicht gesehen wird, für Sensoren besteht Gefahr.
      sensorThreat[i] = sensorMap.GetSensorThreat(i) *
      AREA::THREAT_MULTIPLIER;
    }

    else
    {
      //Wenn die Flughöhe kleiner ist als die maximale Höhe, in der
      //man nicht gesehen wird, für Sensoren besteht keine Gefahr.
      sensorThreat[i] = AREA::iNO_THREAT;
    }

      //Für Waffe
      //Vergleich ob die Flughöhe höher ist als die maximale Höhe, in der
      //man nicht gesehen wird, für Waffen.
    if (fleightHeightMSL >= engagementAreaHeightMSL)
    {
      //Wenn die Flughöhe größer oder gleich der maximalen Höhe, in der
      //man nicht gesehen wird, für Waffen besteht Gefahr.
      engagementThreat[i] = engagementMap.GetEngagementThreat(i) *
      AREA::THREAT_MULTIPLIER;
    }

    else
    {
      //Es besteht keine Gefahr.
      engagementThreat[i] = AREA::iNO_THREAT;
    }
  }
```

Durch die Berechnung des Bedrohungsschattens wird die Abschirmung des Radars hinter Bergen gewährleistet und somit kann das Luftfahrzeug unentdeckt in diesem Bedrohungsschatten fliegen. Der Bedrohungsschatten wird bei der späteren Routensuche als normales Gelände betrachtet.

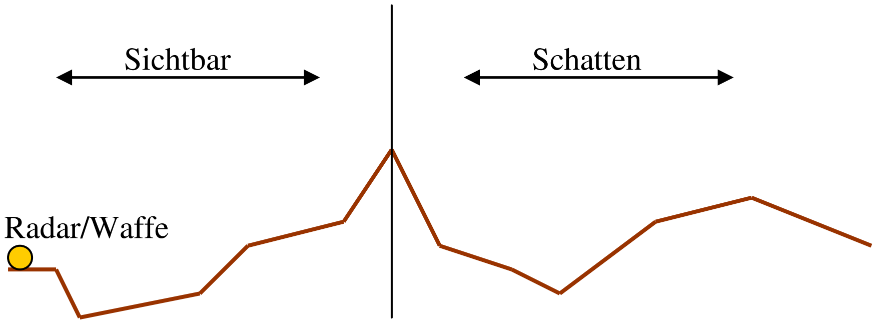


Abbildung 22 Bedrohungsschatten 1

Allerdings ist es möglich, dass nach einem Bedrohungsschatten ein höheres Gelände kommt, das nicht im Schatten liegt. Dies wird dann als bedroht erkannt.

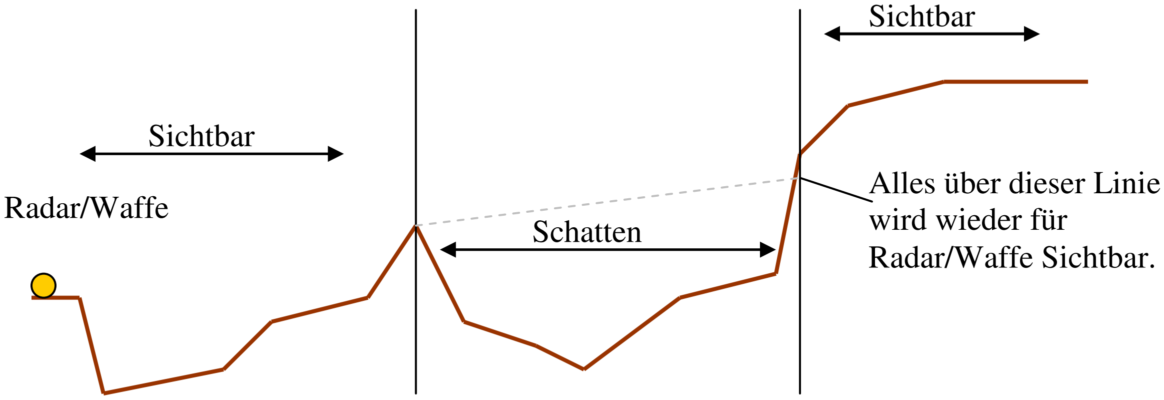


Abbildung 23 Bedrohungsschatten 2

Somit kann durch das Tal völlig unerkannt durchgeflogen werden. Jedoch ist man auf dem Berg wieder sichtbar.

Das folgende Bild zeigt die Anwendung. Radarbedrohungen werden blau, Waffenbedrohungen werden rot dargestellt.

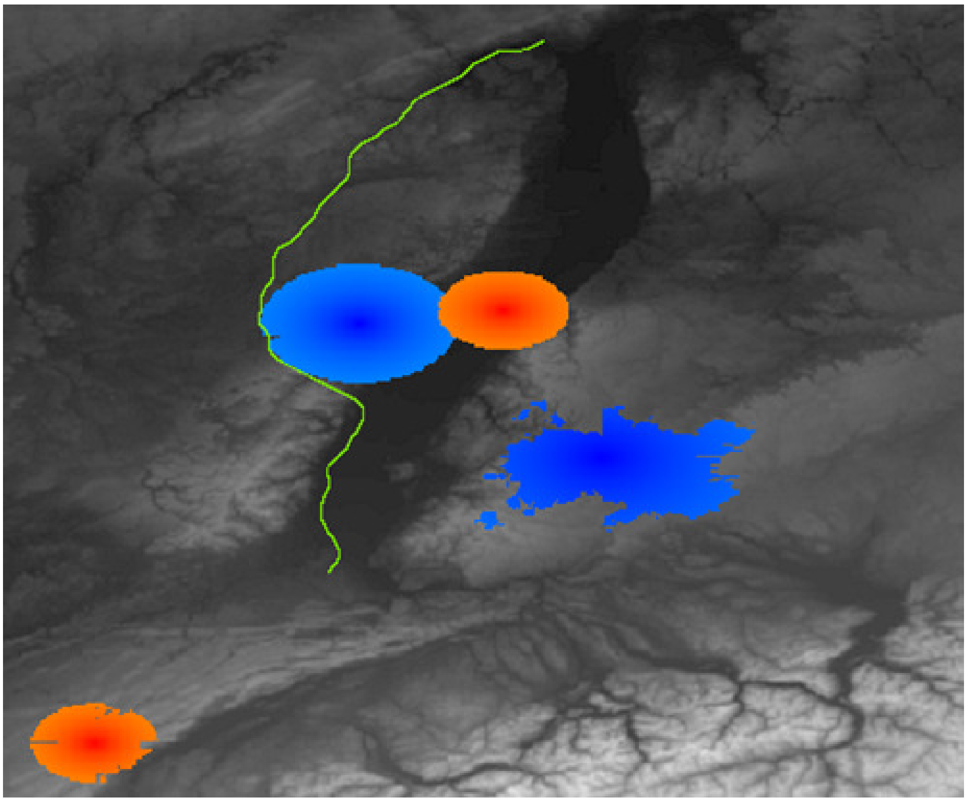


Abbildung 24 Anwendung Radar-/Waffenbedrohung

Wie auf dem Bild zu erkennen ist, werden die Farben nach außen hin schwächer. Dies symbolisiert die Abschwächung der Gefahr. Zudem ist zu erkennen, dass der Bedrohungsschatten als normales Gelände dargestellt ist. Ein Sensor oder eine Waffe sind normalerweise als Kreise dargestellt. Jedoch ändert sich dies durch die Abschattung, welche als normales Gelände dargestellt ist. Sensoren und Waffen können sich überlagern, jedoch wird an den Überlagerungsstellen nur die Waffenbedrohung angezeigt, da sie wesentlich gefährlicher ist als die Sensorbedrohung. Man hat zwei voneinander unabhängige Bedrohungsarten, die nach Wunsch eingestellt und bearbeitet werden können.

6. Optimierung nach Flughöhe

Jeder Knoten im Graph besitzt eine Flughöhe. Nach dieser Flughöhe soll optimiert, sprich die Route mit der geringsten Höhe gefunden werden.

6.1 Höhe als Kantengewichte

Der Ansatz war, die Höhe des Endpunktes einer Kante als Kantengewichte zu benutzen. Durch den Bellman-Ford Algorithmus wird so die Route mit der geringsten Summe der Flughöhen gefunden. Dadurch sind aber zum Beispiel zwei Kanten mit der Höhe 200m schlechter als eine Kante mit der Höhe 300m (Abbildung 25). Die Optimierung sollte nicht die geringste Summe der Flughöhe finden, sondern die Route mit der geringsten Flughöhe unabhängig von der Länge.

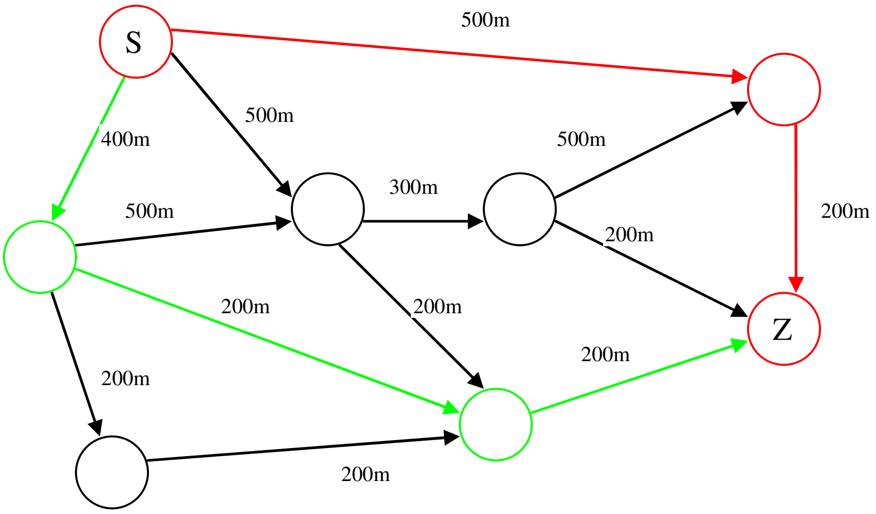


Abbildung 25 Höhe als Kantengewichte

In Abbildung 25 wird die rot markierte Route als günstigste Route gefunden, da die Summe der Kanten mit 700m die geringste ist. Allerdings wird über eine Erhöhung mit 500m Höhe geflogen. Die grün markierte Route ist die Route, welche bei der Suche nach geringster Flughöhe gefunden werden sollte, da sie am flachsten ist.

6.2 Greedy Algorithmus

Ein möglicher Ansatz ist die Verwendung des Greedy-Algorithmus aus der Graphentheorie. Es wird vom Anfangsknoten ausgehend immer die Kante mit der kleinsten Höhe bevorzugt. Dies garantiert aber nicht die Route mit der geringsten Flughöhe. Falls am nächsten Punkt die weiterführenden Kanten wesentlich höher liegen als die weiterführenden Kanten des nicht bevorzugten Weges, bekäme man eine schlechtere Route. Man kann nicht weit genug nach vorne schauen, um dieses Problem zu umgehen ohne dass die Komplexität des Algorithmus zunimmt. Außerdem ist bei gleicher Höhe an verschiedenen Kanten nicht entscheidbar, welche Kante bevorzugt werden soll, da man nicht weiß, welcher Weg der bessere ist.

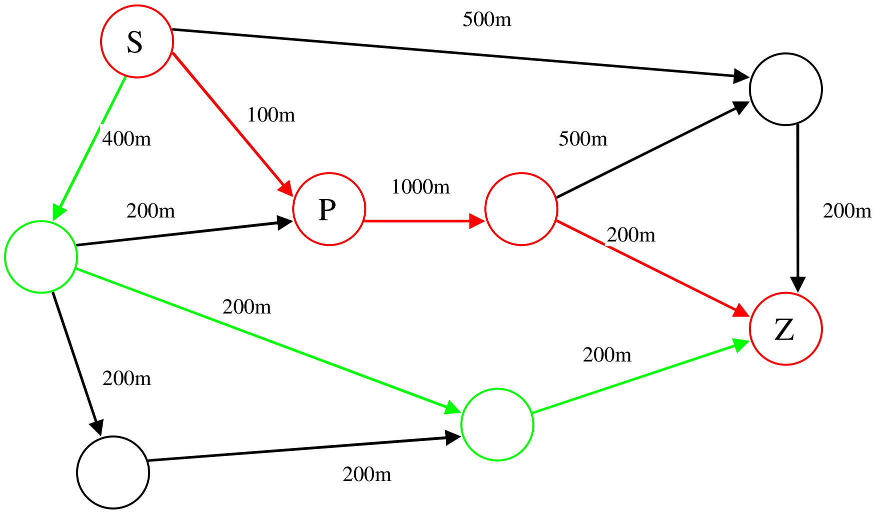


Abbildung 26 Greedy Algorithmus

Die rot markierte Route wird als günstigste Route gefunden, da vom Startknoten S ausgehend immer die Kante mit der geringsten Höhe als günstigste betrachtet wird. Im Punkt P jedoch bleibt keine andere Wahl als über die mit 1000m gewichtete Kante zu gehen. Die grün markierte Route wäre die gewünschte Lösung. Man könnte diesen Algorithmus noch etwas verbessern, indem man ihn nach vorne schauen lässt. So kann eine bessere Lösung gefunden werden, allerdings mit einem hohem Zeitaufwand, da jede Kante des voraus betrachteten Knotens betrachtet werden muss. Allerdings kann diese Situation erst sehr spät auftreten und somit durch das Vorausschauen nicht erkannt werden.

Man dürfte die Tiefe nicht beschränken und würde zur normalen Tiefensuche gelangen; wobei bei der Tiefensuche das Problem besteht, dass man wieder über die Summe der Gewichte zum Zielknoten gelangt und dann wieder dasselbe Problem wie bei 6.1 hat und eine ungünstigere aber kürzere Route findet.
Da im schlimmsten Fall alle möglichen Pfade zu allen möglichen Knoten betrachtet werden müssen, beträgt die Kompläxität von Iterativer Tiefensuche $O(|V|+|E|)$.

6.3 Lösung durch Bildung von Intervallen

Da der Bellman-Ford-Algorithmus die Kantengewichte summiert, ist die Idee, die Gewichte der Kanten mit großer Flughöhe künstlich zu erhöhen. Durch die Erhöhung werden niedrige Kanten bevorzugt.
Die Lösung war die Bildung von Intervallen über die Höhe. Um noch eine stärkere Unterscheidung zu erreichen, werden die Höhen durch einen festen Wert dividiert. Anschließend multipliziert man die Höhen in den einzelnen Intervallen mit einem bestimmten Wert, der dem Intervall zugeordnet ist. Da die Geländehöhen beschränkt sind, ist die Anzahl der Intervalle fest. Nach längerem Testen haben sich Intervalle mit folgenden Werten als beste Lösung herausgestellt.

Kanten im Intervall < 100m bekommen den Wert 10*Höhe/600.
Kanten im Intervall >= 100m und < 250m bekommen den Wert 100*Höhe/550.
Kanten im Intervall >= 250m und < 500m bekommen den Wert 150*Höhe/500.
Kanten im Intervall >= 500m und < 750m bekommen den Wert 1000*Höhe/450.
Kanten im Intervall >= 750m und < 1000m bekommen den Wert 1500*Höhe/250.
Kanten im Intervall >= 1000m und < 1500m bekommen den Wert 10000*Höhe/150.
Kanten im Intervall >= 1500m und < 2000m bekommen den Wert 15000*Höhe/100.
Kanten im Intervall >= 2000m und < 2500m bekommen den Wert 100000*Höhe/50.
Kanten im Intervall >= 3000m bekommen den Wert 1000000*Höhe/50.

Somit werden die Punkte, die im Intervall zwischen 250m-500m liegen, 10-fach so stark gewichtet als die im Intervall <250m liegenden. Die Höhe wird dazu multipliziert, so dass in den Intervallen selbst noch leicht unterschieden wird. Das führt dazu, dass zuerst 10 Kanten im Intervall <250 bevorzugt werden, bevor eine Kante im Intervall 250m-500m bevorzugt wird. Dadurch wird eine günstige Route gefunden. Jedoch wird nicht garantiert, dass es die Route mit der geringsten Flughöhe unabhängig von der Länge ist. Es wird davon ausgegangen, dass die Route mit der geringsten Flughöhe wesentlich länger sein kann, aber dann auch uninteressant für die Praxis wird.
Die Intervalle werden über die an den Kanten gespeicherten Höhen gebildet.

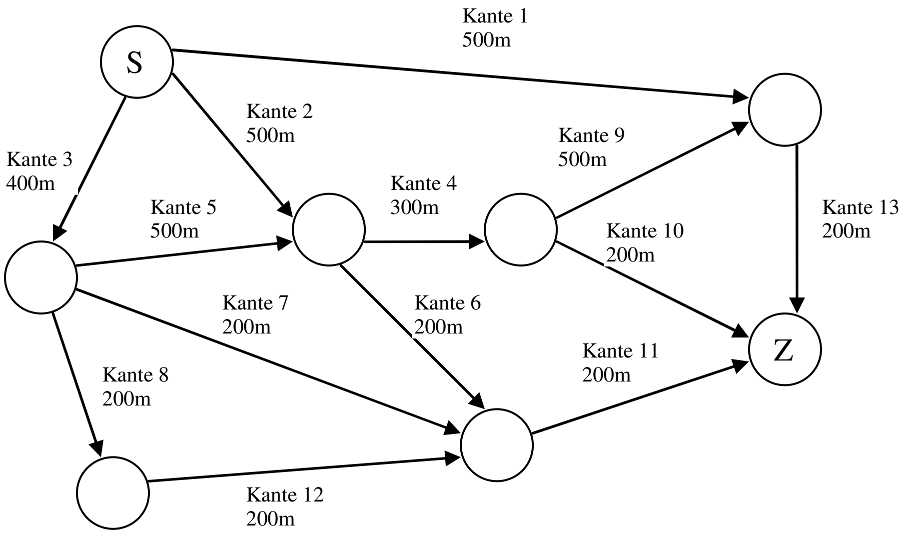


Abbildung 27 Bildung von Intervalle Höhe an die Kanten gespeichert

Berechnung der Intervalle:

Kante	Höhe	Intervall	Rechnung	Wert
1	500m	>= 500m < 750m	1000*500/450	1111,11
2	500m	>= 500m < 750m	1000*500/450	1111,11
3	400m	>= 250m < 500m	150*400/500	120
4	300m	>= 250m < 500m	150*300/500	90
5	500m	>= 500m < 750m	1000*500/450	1111,11
6	200m	>= 100m < 250m	100*200/550	36,36
7	200m	>= 100m < 250m	100*200/550	36,36
8	200m	>= 100m < 250m	100*200/550	36,36
9	500m	>= 500m < 750m	1000*500/450	1111,11
10	200m	>= 100m < 250m	100*200/550	36,36
11	200m	>= 100m < 250m	100*200/550	36,36
12	200m	>= 100m < 250m	100*200/550	36,36
13	200m	>= 100m < 250m	100*200/550	36,36

Nach der Bildung der Intervalle wird der errechnete Wert an den entsprechenden Kanten gespeichert. (siehe Abbildung 28)

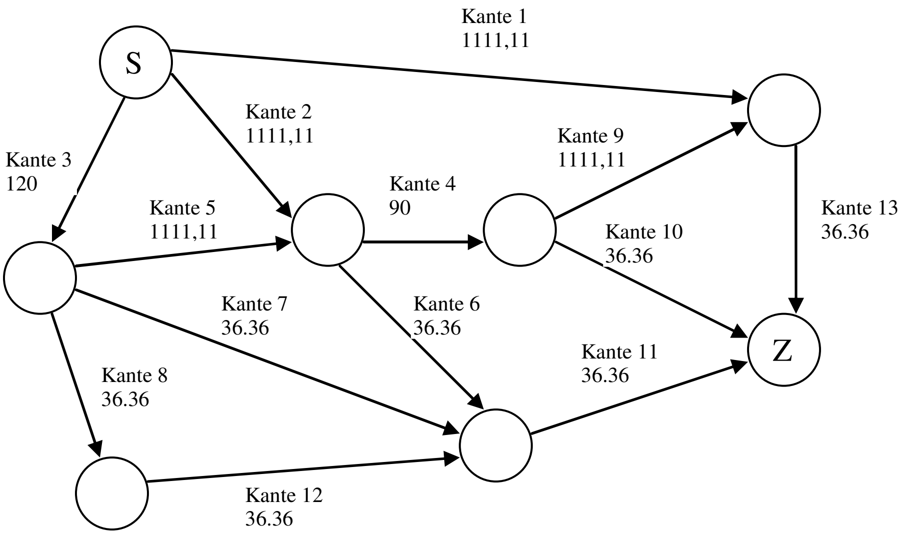


Abbildung 28 Bildung von Intervalle berechneter Wert speichern

Nachdem die Kanten nun bewertet sind, wird der Bellman-Ford-Algorithmus darauf angewendet und die günstigste Route berechnet.

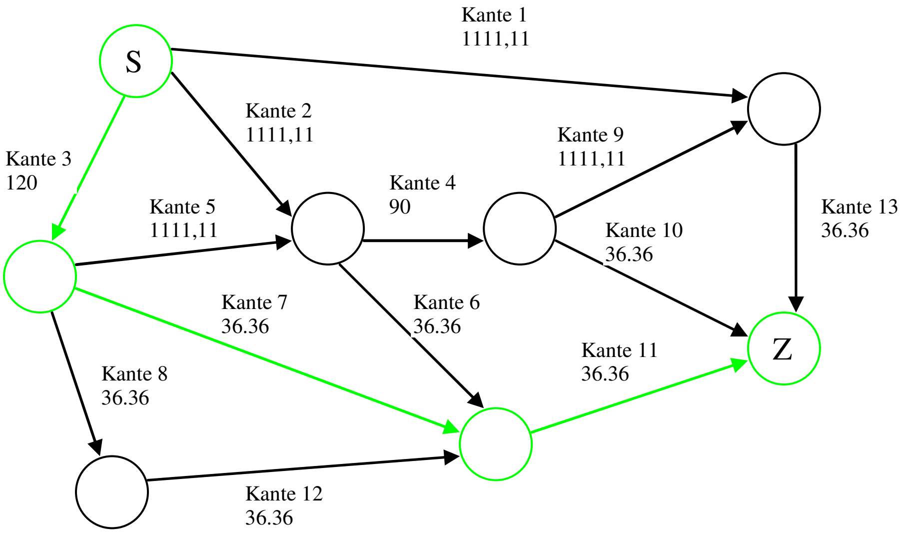


Abbildung 29 Bildung von Intervalle günstigste Route

Die grün markierte Route wird als günstigste Route gefunden. Jedoch kann es vorkommen, dass es eine bessere Route gibt, wenn es so viele flache Kanten gibt dass dadurch eine etwas höhere Kante besser werden kann. Jedoch ist diese bessere Route wesentlich länger als die gefundene Route und für die Praxis uninteressant.

Code:

```
for jeden Knoten
        for jede ausgehende Kante
                if höhe in Intervall <100
                        Gewicht = höhe/600*10
                if höhe in Intervall >= 100m und < 250m
                        Gewicht = höhe/550*100
                if höhe in Intervall >= 250m und < 500m
                        Gewicht = höhe/500*150
                if höhe in Intervall >= 500m und < 750m
                        Gewicht = höhe/450*1000
                if höhe in Intervall >= 750m und < 1000m
                        Gewicht = höhe/250*1500
                if höhe in Intervall >= 1000m und < 1500m
                        Gewicht = höhe/150*10000
                if höhe in Intervall >= 1500m und < 2000m
                        Gewicht = höhe/100*15000
                if höhe in Intervall >= 2000m und < 2500m
                        Gewicht = höhe/50*100000
                if höhe in Intervall >= 3000m
                        Gewicht = höhe/50*1000000
```

7. Optimierung nach Steig- und Sinkrate

7.1 Problemstellung

Ein Luftfahrzeug besitzt eine maximale Steig- und Sinkrate. Die Steig- und Sinkrate beschreiben, welche Höhendifferenz ein Luftfahrzeug in einer bestimmten Zeit überwinden kann. Die maximale Steig- und Sinkrate können dabei unterschiedlich sein. Da das Gelände nicht überall fliegbar ist, wird die Flughöhe so angepasst, dass die Höhenunterschiede zwischen einem Punkt und seinen Nachbarpunkten mit der maximalen Steig- und Sinkrate fliegbar sind. Es wird von einer fliegbaren Route gesprochen, sobald alle Höhenunterschiede der Route mit den Flugleistungsparametern des zugrundeliegenden Luftfahrzeugs bewältigt werden können. Die Fliegbarkeit wird bei Steigungen durch früheres Steigen erreicht. Bei einem Gefälle wird die Fliegbarkeit durch verzögertes Sinken erreicht. Durch das frühere Steigen und verzögerte Sinken entsteht an den betroffenen Punkten eine größere Flughöhe. Dadurch können diese Punkte in einen anderen Bedrohungsbereich fallen. Durch diese Bedrohungsänderung an den Punkten kann nun eine andere Route wesentlich günstiger werden als die aktuelle Route. Um die günstigste Route gewährleisten zu können, müssen alle Höhen, die geändert werden, vor der Berechnung der günstigsten Route bekannt sein. Dazu wird vor der Berechnung der günstigsten Route die komplette Landschaft den Flugleistungsparametern des zugrundeliegenden Luftfahrzeugs angepasst.

7.2 Berechnung der optimalen Höhe

Um die Höhe den Flugleistungsparametern anzupassen, müssen Höhen geändert werden. Diese müssen so geändert werden, dass die Steigung nicht zu groß für die Flugleistungsparameter ist. Man unterscheidet zwei Arten von Steigungen: zum einen das Gefälle und zum anderen den Anstieg. Beide müssen unterschiedlich behandelt werden, da die Steig- und Sinkrate unterschiedlich sein können.

Bei einem Anstieg sieht das Steigungsdreieck wie folgt aus:

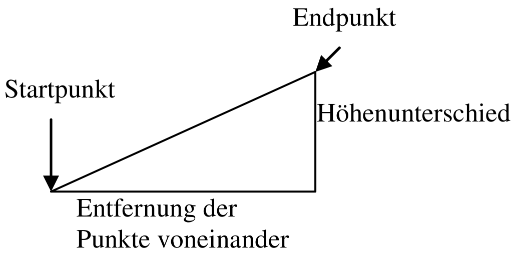


Abbildung 30 Positives Steigungsdreieck

Man hat nur eine Möglichkeit die Steigung zu verringern. Das Absenken des Endpunktes kommt nicht in Frage, da man sonst in das Gelände hinein fliegen würde. Somit bleibt nur die Option, den Startpunkt anzuheben. Um die optimale Höhe des Startpunktes zu ermitteln, müssen drei Variablen bekannt sein: der Höhenunterschied oder die Höhen beider Punkte, die Entfernung der Punkte voneinander und die Steigung. Die Steigrate muss für die Steigung umgerechnet werden. Um die Steigrate umzurechnen wird noch die Bodengeschwindigkeit (groundspeed) des Flugkörpers benötigt. Die Steigung wird dann wie folgt berechnet:

$$Steigung = Steigrate(m/s)/Geschwindigkeit(m/S)$$

Die Berechnung der optimalen Höhe des Startpunktes sieht dann wie folgt aus:

$$OptimaleHöhe = -(Steigung * Entfernung) + HöheEndpunkt$$

Bei einem Gefälle sieht das Steigungsdreieck wie folgt aus:

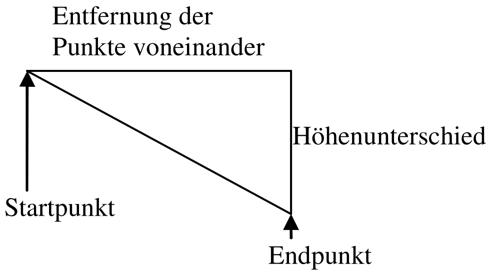


Abbildung 31 Negatives Steigungsdreieck

Bei einem Gefälle ist die Steigung negativ. Man hat wieder nur eine Möglichkeit die Steigung zu verringern. Das Absenken des Startpunktes kommt nicht in Frage, da man sonst in das Gelände hinein fliegen würde. Nun bleibt wieder nur eine Option übrig: den Endpunkt anheben. Für die Berechnung der optimalen Höhe sind dieselben Bedingungen zu beachten wie bei einer Steigung. Die Berechnung der benötigten Steigung sieht wie folg aus:

$$Steigung = -Sinkrate(m/s)/Geschwindigkeit(m/S)$$

Da in dieser Arbeit die Sinkrate auch positiv ist, muss sie negiert werden. Denn die Steigung im Steigungsdreieck muss negativ sein.

Die Berechnung der optimalen Höhe des Endpunkts sieht dann wie folgt aus:

$$OptimaleHöhe = -(Steigung * Entfernung) + HöheStartpunkt$$

7.3 Ansätze

7.3.1 Rekursive Anpassung

Der erste Ansatz war die rekursive Anpassung der Höhe vom Startknoten aus. Die Idee war, vom Startknoten ausgehend, alle Nachbarknoten zu betrachten und die Höhe so anzupassen, dass sie fliegbar ist. Bei einem zu starken Anstieg muss der aktuelle Knoten angepasst werden. Dies wird durch das Verschieben des Knotens nach oben erreicht. Bei einem zu starken Gefälle muss der Nachfolgeknoten angepasst werden. Dies geschieht durch das Verschieben des Nachfolgeknotens nach oben. Jedoch kann es nach dem Verschieben eines Knotens sein, dass seine Höhe zu hoch für seine Vorgänger oder Nachfolger wird. Da dies passieren kann, müssen bei einer Änderung der Höhe alle seine Vorgänger und seine Nachfolger, aber nur die schon betrachtet wurden, erneut betrachtet werden. Sollte eine Kante nicht fliegbar sein, muss der Vorgänger oder Nachfolger, durch Verschieben des Punktes nach oben, angepasst werden.

Nun müssen alle seine Nachfolger und Vorgänger betrachtet und erneut angepasst werden. Sollte sich ein Nachfolger oder Vorgänger ändern, müssen auch seine Nachfolger und Vorgänger angeschaut und angepasst werden. Dies muss so lange gemacht werden, bis sich nichts mehr ändert. Das Ganze ist natürlich sehr zeitaufwendig und dadurch wird der Algorithmus auch dementsprechend langsam.

Die Idee wird anhand eines Beispiels verdeutlicht. In diesem Beispiel nimmt man an, dass das Luftfahrzeug entlang einer Kante 100m Höhenunterschied überbrücken kann, sowohl steigend als auch sinkend.

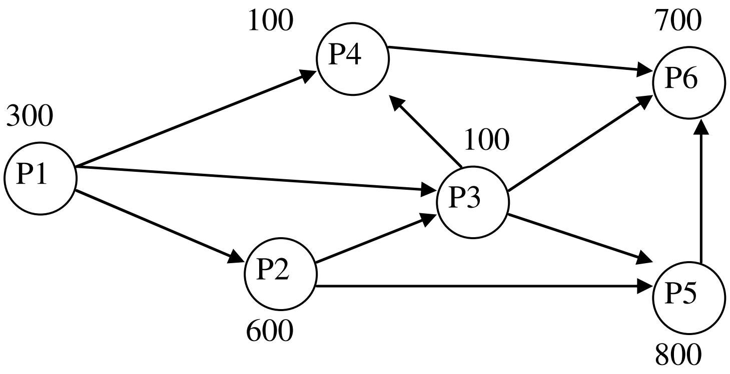


Abbildung 32 Rekursive Anpassung Ausgangssituation

Die Abbildung 32 zeigt die Ausgangssituation des Graphen, welcher angepasst werden soll. Es handelt sich um einen gerichteten azyklischen Graphen mit sechs Knoten und zehn Kanten. Nun sollen die Höhen wie oben beschrieben angepasst werden. Dabei können nicht mehr als 100m Höhenunterschied überbrückt werden. Im ersten Schritt werden alle ausgehenden Kanten des Knotens mit keinem Vorgänger (P1) betrachtet und angepasst.

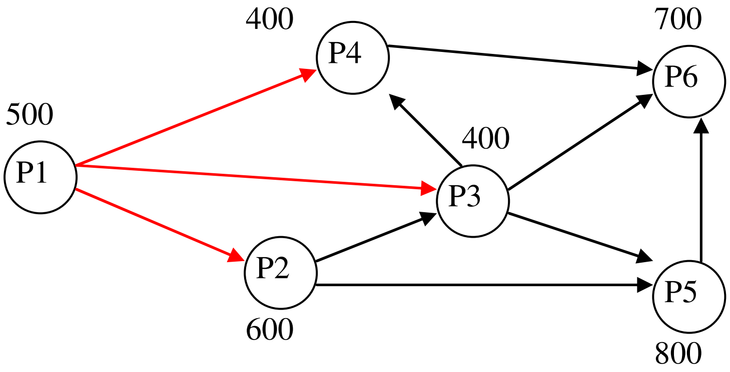


Abbildung 33 Rekursive Anpassung erster Schritt

Der Anfangspunkt muss aufgrund des Knotens P2 auf 500m angehoben werden. Somit müssen die Knoten P3 und P4 von 100m auf 400m angehoben werden. Im nächsten Schritt werden alle Nachfolger des Knotens P2 betrachtet und die Höhen angepasst.

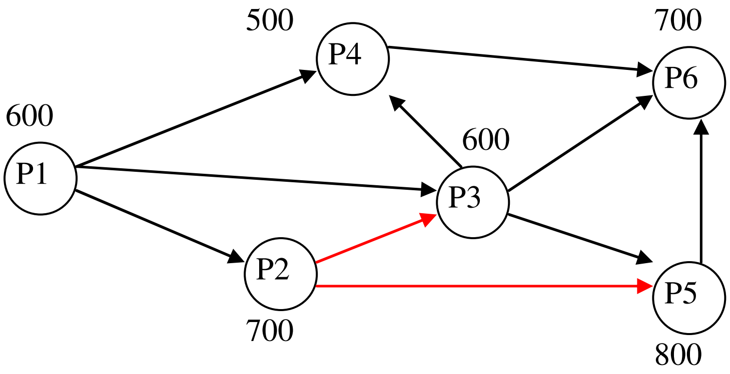


Abbildung 34 Rekursive Anpassung zweiter Schritt

Aufgrund des Knotens P5 muss der Knoten P2 auf 700m angehoben werden. Dadurch müssen alle Vorgänger des Knotens P2 überprüft werden. Der Knoten P1 wird dementsprechend auf 600m angehoben. Es müssen alle Nachfolger und Vorgänger von Knoten P1 betrachtet und an die neue Höhe angepasst werden. Dadurch wird die Höhe von Knoten P4 auf 500m angehoben.
Im nächsten Schritt werden alle Nachfolger von Knoten P3 betrachtet und die Höhen dementsprechend angepasst.

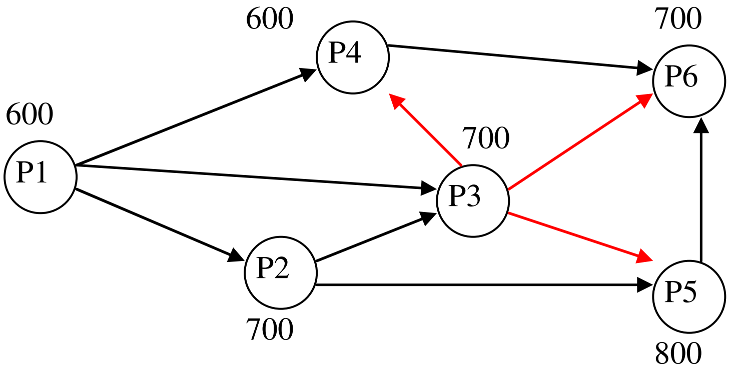


Abbildung 35 Rekursive Anpassung dritter Schritt

Die Höhe von Knoten P3 wird aufgrund des Knotens P5 auf 700m angehoben. Der Knoten P6 bleibt ohne Auswirkungen; der Knoten P4 jedoch muss auf 600m angehoben werden. Nun werden alle Nachfolger und Vorgänger von Knoten P4 betrachtet, jedoch ohne Auswirkungen. In den nächsten Schritten werden die Knoten P4, P5 und P6 betrachtet, welche aber keine Auswirkungen mehr haben, da all ihre Kanten fliegbar sind.

Der fertig angepasste Graph sieht wie folgt aus. Es ist jede Kante fliegbar, jedoch ist der Zeitaufwand viel zu groß.

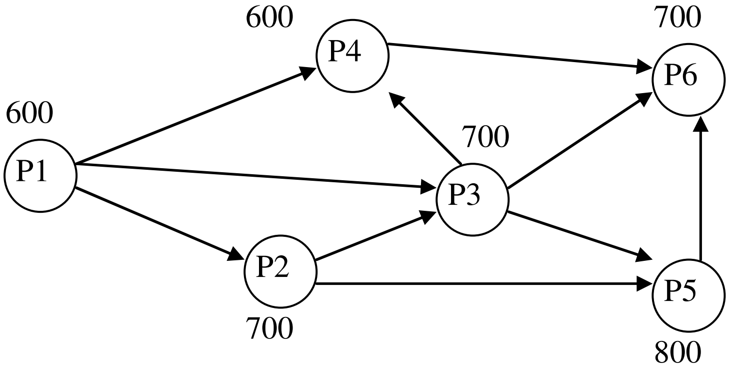


Abbildung 36 Rekursive Anpassung komplette Anpassung

Dieser Algorithmus wurde in dieser Diplomarbeit nicht implementiert, sondern nur theoretisch betrachtet. Der Aufwand ist exponentiell. Da die Laufzeit auf großen Graphen viel zu groß ist, kam er nicht in Betracht.

7.3.2 Geglättete Landschaft über die 16 Nachbarpunkte

Eine weitere Idee war, die komplette Landschaft zu glätten. Das heißt, dass der Reihe nach über die einzelnen Punkte im Raster gegangen wird und jeder Punkt an das Maximum seiner 16 Nachbarn anpasst und anschließend die zu tief liegenden Punkte nach oben zieht. Dies wird der Reihe nach für jeden Punkt gemacht. Wenn der letzte Punkt angepasst wurde, wird von neuem begonnen. Somit wird sichergestellt, dass durch die Verschiebung von einzelnen Punkten kein neues Maximum entsteht, welches nicht mehr fliegbar wäre. Dies wird so oft wiederholt, bis es keine Änderungen mehr gibt. Beim Anpassen der Punkte im Raster ist zu beachten, dass Punkte, welche am Rand liegen, weniger als 16 Nachbarn haben. Die Abbildung 37 zeigt, wie der Reihe nach vorgegangen wird, um jeden Punkt anzupassen. Zudem wird dargestellt, welche 16 Nachbarn gemeint sind.

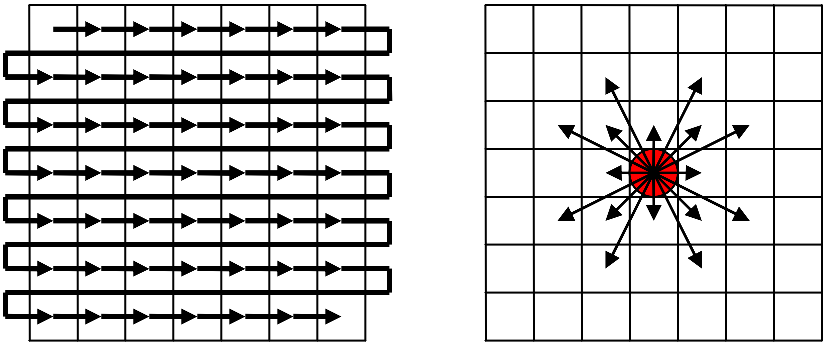


Abbildung 37 Reihenfolge und Nachbarn

Code:

```
changes=1
while(changes!=0)
        changes=0
        for jeden Punkt im Gitter
                Get alle Nachbarn
                Finde Maximum

                if Maximum nicht fliegbar
                        passe Punkt an.
                        passe alle zu tief liegende Nachbarn an.
                        changes++
```

Die Komplexität des Algorithmus ist im Best Case linear von der Anzahl der Punkte und Anzahl der Nachbarn im Raster abhängig, da genau einmal über das gesamte Raster gegangen und jeder Punkt genau einmal angepasst wird. Zudem ist das Finden des Maximums und Anpassen der Höhe mit einem konstanten Aufwand möglich. Außerdem entspricht die Anzahl der Punkte im Raster der Anzahl der Knoten. Somit ist der Aufwand linear abhängig von der Anzahl der Knoten. Und da die Anzahl der Nachbarn 16 nicht übersteigt, ist die Anzahl der Kanten linear von der Anzahl der Knoten abhängig. Somit kann der Aufwand geschätzt werden auf:

$$O(|V|)$$

Im Worst Case ist die Komplexität n^2, da mindestens ein Punkt nach jedem Durchlauf an das Maximum angepasst ist und sich dadurch nicht mehr ändert. Es kann keine Endlosschleife entstehen, da im Extremfall die Landschaft zu einer glatten Fläche gemacht wird, welche die Höhe des Gelände-Maximums besitzt. Dadurch ist gewährleistet, dass eine Lösung nach endlicher Zeit gefunden wird. Die Komplexität berechnet sich wie folgt:

$$n+(n-1)+(n-2)+...+1=\frac{n(n+1)}{2}$$

$$\lim_{n\to\infty} O(n^2)$$

Für die Größe der verwendeten Graphen ist dieser Aufwand akzeptabel. Zudem wird durch das Nachziehen aller zu tief liegenden Nachbarn eine Performance-Steigerung erzielt, da dadurch das Entstehen von neuen Maxima verringert wird und somit muss das Raster nicht so oft durchlaufen werden.

Jedoch wird durch diesen Algorithmus das Gelände sehr stark geglättet und somit eine wesentlich schlechtere Route gefunden. Die Vereinfachung ist zu stark und somit wird die Route zu schlecht. Aus diesem Grund wurde diese Variante nicht als Lösung genommen.

7.3.3 Geglättete Landschaft über den Graphen

Die Idee ist, einen ähnlichen Algorithmus wie den in 7.3.2 beschriebenen, nicht auf die Punkte im Raster anzuwenden, sondern auf den gerichteten azyklischen Graph. Durch die Bestimmung der Kreisfreiheit können Knoten mit Eingangskanten keine 16 Nachbarn haben, da sonst ein Zyklus entstehen würde.

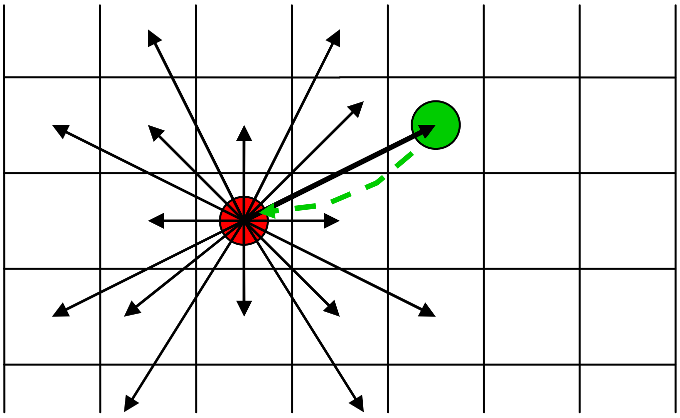


Abbildung 38 Entstehen eines Zyklus

Die Abbildung 38 veranschaulicht dies, da der grüne Punkt bereits eine Eingangskante besitzt, er kann somit keine 16 Ausgangskanten haben. Wenn er trotzdem 16 Ausgangskanten hätte, würde ein Zyklus entstehen, wie der grün gestrichelte Pfeil andeutet. Durch den Zwang der Kreisfreiheit besitzt nur der Ursprung 16 Nachbarknoten, da er keine Eingangskante besitzt. Berücksichtigt man nun, dass ein Knoten mehrere Eingangskanten besitzen kann, so ergibt sich ein mittlerer Verzweigungsfaktor von ca. acht.

Dadurch wird die Berechnung des Maximums nicht auf 16 sondern nur noch auf durchschnittlich acht Nachbarn durchgeführt. Somit wird im Durchschnitt nur halb so stark vereinfacht.

Alle Knoten werden der Reihe nach abgearbeitet und jeder Nachbar, der durch eine Ausgangskante erreicht werden kann, betrachtet, ob der Höhenunterschied fliegbar ist oder nicht. Bei einer zu großen Steigung wird der aktuelle Knoten so angepasst, dass der Nachbarknoten erreichbar ist.

Die Höhe stellt sich so automatisch auf das Maximum der Nachbarknoten ein. Bei einem zu starken Gefälle wird der Nachbarknoten angehoben. Jedoch wird nicht wie in 7.3.2 zuerst das Maximum berechnet, um die zu tiefen Nachbarknoten nachzuziehen, sondern es werden die Nachbarn der Reihe nach geprüft, ob sie zu hoch oder zu tief liegen. Nachdem der komplette Graph durchlaufen wurde, wird von neuem begonnen, da durch die Änderungen neue Maxima entstehen können, die nicht fliegbar sind.

Das folgende Beispiel zeigt die Arbeitsweise des Algorithmus. Man nimmt an, dass das Luftfahrzeug maximal 100 Höhenmeter pro Kante steigen und sinken kann. Die Abbildung 39 zeigt den Graph in seiner Ausgangssituation.

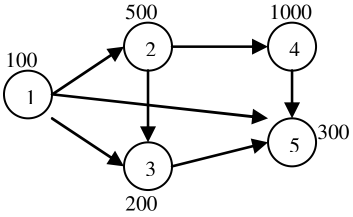


Abbildung 39 Geglättete Landschaft über den Graph Ausgangssituation

Im ersten Schritt werden alle Nachbarn des Knoten 1, die durch eine Ausgangskante erreicht werden, betrachtet und bearbeitet. Dabei wird der Reihe nach überprüft, ob die Ausgangskante fliegbar ist oder nicht.

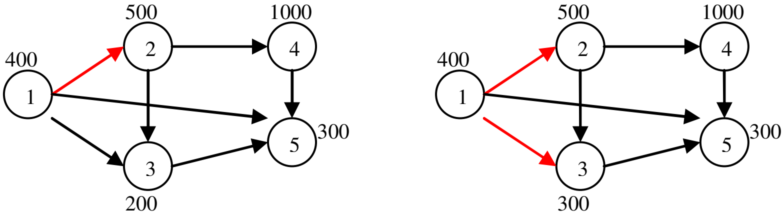


Abbildung 40 Geglättete Landschaft über den Graph erster Schritt

Zuerst wird die Ausgangskante von Knoten 1 zu Knoten 2 betrachtet. Der Höhenunterschied ist höher als 100m, also ist die Kante nicht fliegbar und muss angepasst werden. Um sie fliegbar zu machen, muss der Knoten 1 auf 400m hoch gesetzt werden. Danach wird die nächste Ausgangskante betrachtet. Der Höhenunterschied ist höher als 100m. Um die Ausgangskante fliegbar zu machen, muss der Knoten 3 auf 300m hoch gesetzt werden.

Da alle Ausgangskanten betrachtet wurden, wird zum nächsten Knoten weiter gegangen. Es handelt sich hierbei um Knoten 2.

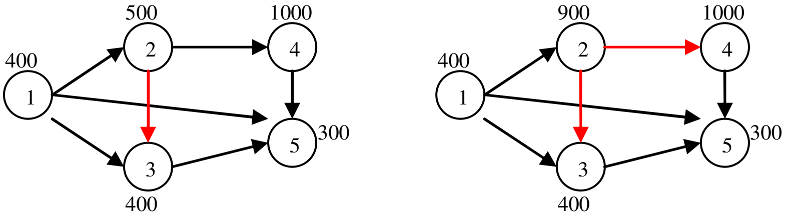


Abbildung 41 Geglättete Landschaft über den Graph Knoten 2

Als erste Ausgangskante wird die Kante von Knoten 2 zu Knoten 3 betrachtet. Dabei wird festgestellt, dass diese nicht fliegbar ist. Um sie fliegbar zu machen, muss der Knoten 3 auf 400m hoch gesetzt werden. Als nächstes wird die Kante zwischen Knoten 2 und Knoten 4 betrachtet. Um diese fliegbar zu machen, muss der Knoten 2 auf 900m hoch gesetzt werden. Dabei wird die Kante zwischen Knoten 2 und Knoten 3 nicht fliegbar, jedoch wurde diese schon zuvor betrachtet. Nachdem Knoten 2 vollständig betrachtet wurde, wird Knoten 3 betrachtet.

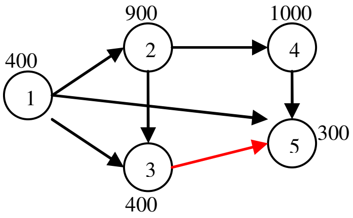


Abbildung 42 Geglättete Landschaft über den Graph Knoten 3

Die Kante zwischen Knoten 3 und Knoten 5 ist fliegbar und somit muss nichts geändert werden. Es wird weiter zu Knoten 4 gegangen und seine Nachbarn betrachtet.

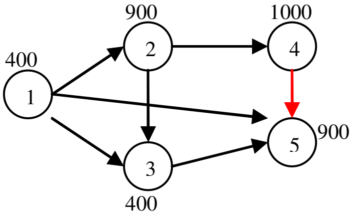


Abbildung 43 Geglättete Landschaft über den Graph Knoten 4

Um die Kante von Knoten 4 zu Knoten 5 fliegbar zu machen, muss Knoten 5 auf 900m hoch gesetzt werden. Knoten 5 muss nicht betrachtet werden, da er keine Ausgangskanten besitzt.

Jetzt wird von vorne begonnen, da neue Maxima entstanden sind. Es wird jeder Knoten von neuem betrachtet. (siehe Abbildung 44)

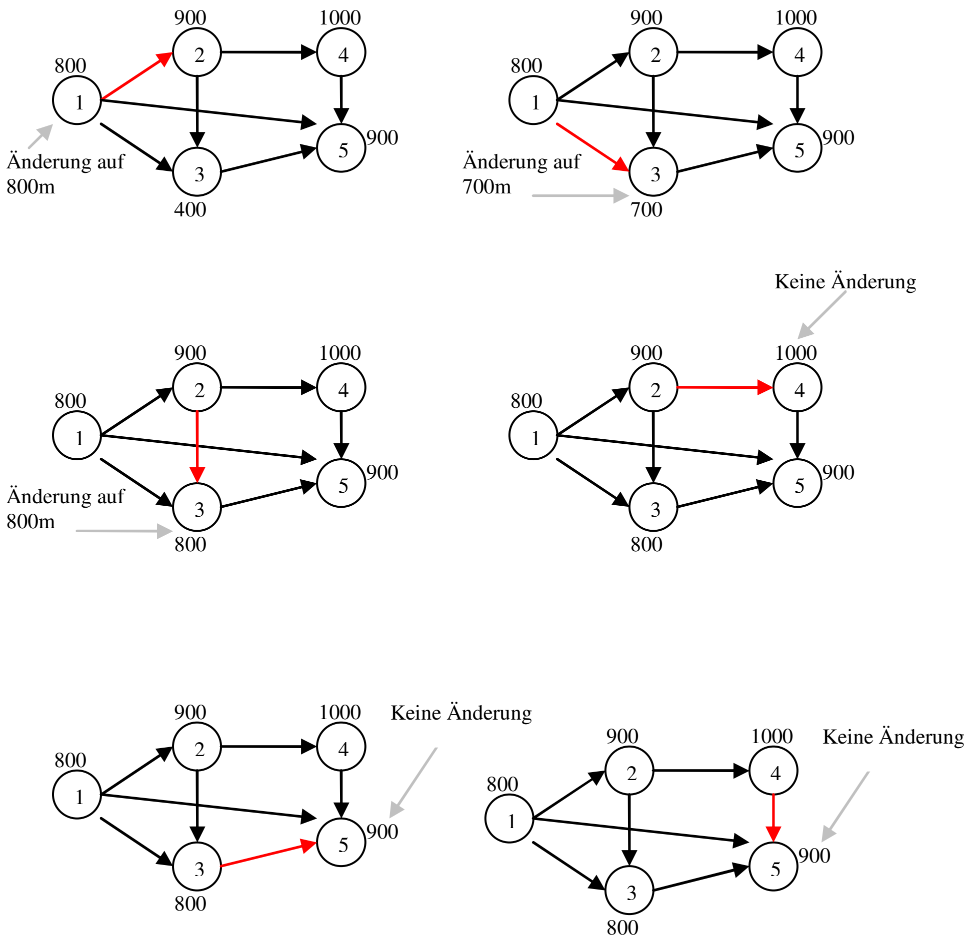


Abbildung 44 Geglättete Landschaft über den Graph erneute Anpassung

Nach diesem erneuten Durchlauf ist der Graph so angepasst, dass jede Kante fliegbar ist. Jedoch muss noch einmal der gesamte Graph durchlaufen werden, um sicherzustellen, dass nicht noch etwas geändert werden muss. Im Praxisfall wird der Graph in der Regel öfters als zweimal durchlaufen werden.

Code:

```
changes=1
while(changes!=0)
        changes=0
        for jeden Knoten
                for jede Ausgehende Kante
                        if  Steigung>Steigrate
                                Anpassen der Höhe des Knotens
                                changes++
                        if Steigung<Sinkrate
                                Anpassen der Höhe des Nachfolger
                                changes++
```

Nach der kompletten Anpassung des Graphen ist die Landschaft so verändert, dass jede Ausgangskante eines jeden Knotens fliegbar ist. Nun kann auf dieser geglätteten Landschaft die günstigste Route berechnet werden. Die Abbildung 45 zeigt im Querschnitt, wie die angepasste Landschaft aussieht.

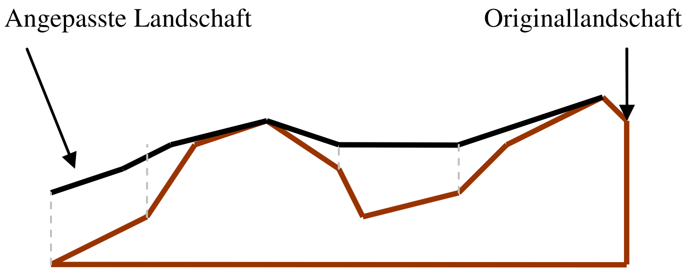


Abbildung 45 Querschnitt der angepassten Landschaft

Die Originallandschaft wird im Querschnitt braun dargestellt. In dieser Landschaft gibt es Steigungen, die nicht fliegbar sind. Die angepasste Landschaft im Querschnitt wird schwarz dargestellt. In dieser Landschaft gibt es keine Steigungen, die nicht fliegbar sind.

Die Komplexität des Algorithmus ist im Best Case linear von der Anzahl der Knoten abhängig, da genau einmal über den gesamten Graphen gegangen und jeder Knoten einmal überprüft wird. Da die Anzahl der Kanten 16 nicht übersteigt und somit die Anzahl der Kanten konstant ist, kann die Komplexität geschätzt werden auf:

$$O(|V|)$$

Im Worst Case ist die Komplexität n^2, da mindest ein Punkt nach jedem Durchlauf an das Maximum angepasst ist und sich dadurch nicht mehr ändert. Es kann keine Endlosschleife entstehen, da im Extremfall die Landschaft zu einer glatten Fläche gemacht wird, welche die Höhe des Gelände Maximums besitzt. Dadurch ist gewährleistet, dass eine Lösung nach endlicher Zeit gefunden wird. Die Komplexität berechnet sich wie folgt:

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$$

$$\lim_{n \to \infty} O(n^2)$$

Für die Größe der verwendeten Graphen ist dieser Aufwand akzeptabel.

Der Algorithmus vereinfacht das Gelände nur halb so stark wie der Algorithmus von 7.3.2, da die Höhe im Mittel nicht an 16 Nachbarn, sondern nur 8 Nachbarn angepasst wird. Dadurch wird eine bessere Route gefunden als bei 7.3.2 und die Komplexität ist nahezu gleich. Allerdings ist die Vereinfachung immer noch sehr stark und kann noch verbessert werden.

7.3.4 Geglättete Landschaft mit Berücksichtigung der Flugrichtung

Die Idee ist, eine geglättete Landschaft zu erstellen unter Berücksichtigung der Flugrichtung. Es werden an allen Kanten die Höhen ihrer Startknoten gespeichert. Dies geschieht mit einer linearen Komplexität, welche von der Anzahl der Kanten abhängt $O(|E|)$.

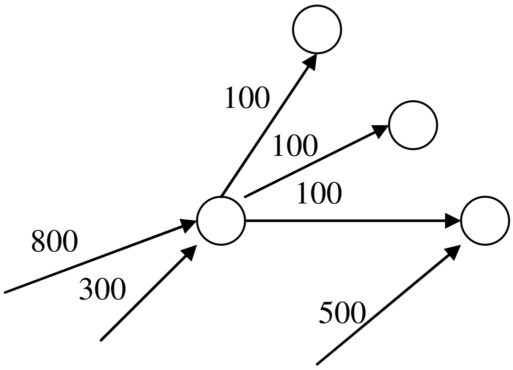


Abbildung 46 gerichteter azyklischer Graph mit Höhen an den Kanten

Die Abbildung 46 zeigt den neuen gerichteten azyklischen Graphen, an dem die Höhen nicht an den Punkten, sondern an den Kanten gespeichert sind. Es sind immer die Höhen des Startknotens an den Kanten gespeichert.

Auf diesem neuen gerichteten azyklischen Graph wird nun der Algorithmus angewendet. Dieser Graph wird der Reihe nach durchlaufen und jeder Knoten betrachtet. Dabei werden die Ausgangskanten der Reihe nach betrachtet und für jede der Endknoten ermittelt.

Eine Kante hat immer eine bestimmte Richtung. Diese Richtung wird für die aktuelle Ausgangskante ermittelt. Zudem werden die Richtungen aller Ausgangskanten des Endknotens bestimmt. Dies hat folgenden Hintergrund: Ein Luftfahrzeug kann nur bestimmte Kursänderungen durchführen. Somit kann man nur bestimmte Ausgangskanten für bestimmte Eingangskanten fliegen. So kann man die Ausgangskanten beschränken, welche für einen Endknoten betrachtet werden sollen.

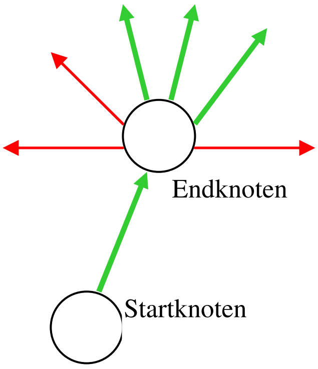


Abbildung 47 Mögliche Ausgangskanten für eine Eingangskante

Die Abbildung 47 beschreibt die möglichen Ausgangskanten des Endpunktes. Grün werden die Kanten dargestellt, deren Richtungsänderung für diese Eingangskante möglich ist. Rot werden die Kanten dargestellt, deren Richtungsänderung für diese Eingangskante nicht möglich ist.

Nachdem die möglichen Kanten ermittelt wurden, wird nun überprüft, ob die Höhenänderung auch fliegbar ist. Sollte die Höhenänderung nicht fliegbar sein, so wird die Höhe angepasst und an der entsprechenden Kante gespeichert. Dies wird für den gesamten Graph gemacht. Nachdem der Graph komplett durchlaufen wurde, wird von neuem begonnen, da neue Höhen, die nicht fliegbar sind, entstanden sein können. Dies wird so oft wiederholt, bis der komplette Graph fliegbar ist.

Diese Vorgehensweise wird an einem Beispiel erläutert. In diesem Beispiel kann das Luftfahrzeug 100m Höhenunterschied, sowohl steigend als auch sinkend, überbrücken.

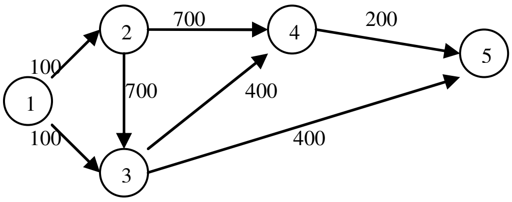


Abbildung 48 Geglättete Landschaft mit Berücksichtigung der Flugrichtung Ausgangssituation

Die Abbildung 48 zeigt den gerichteten azyklischen Graph in seiner Ausgangssituation. Der Graph besteht aus 5 Knoten und 7 Kanten.

Im ersten Schritt werden die Ausgangskanten des Knotens 1 nacheinander betrachtet (siehe Abbildung 49). Zudem wird die Fliegbarkeit der Ausgangskanten des jeweiligen Nachbarknotens überprüft. Anschließend werden die Höhenunterschiede auf ihre Fliegbarkeit überprüft. Sollten sie nicht fliegbar sein, werden die Höhen angepasst und die angepasste Höhe an die entsprechende Kante gespeichert.

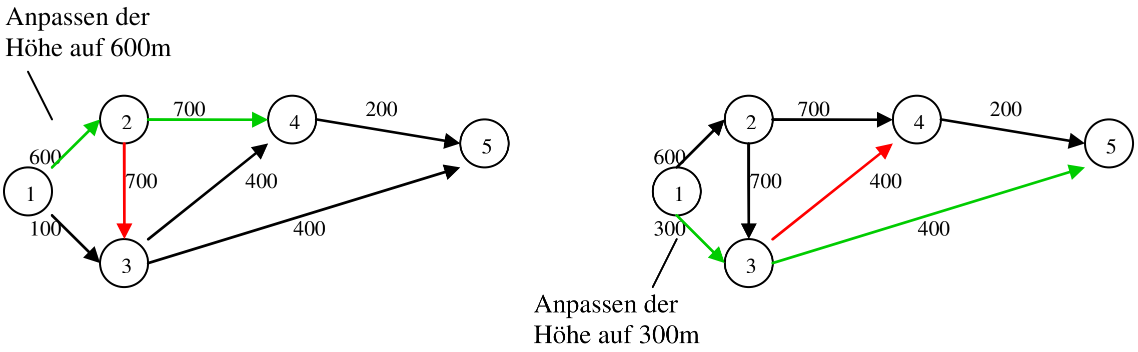


Abbildung 49 Geglättete Landschaft mit Berücksichtigung der Flugrichtung Knoten 1

Kanten, deren Richtungsänderung fliegbar ist, werden grün dargestellt. Kanten die nicht fliegbar sind werden rot dargestellt. Sie werden nicht betrachtet.
Dieser Vorgang wird für jeden Knoten im Graphen wiederholt. (siehe Abbildung 50)

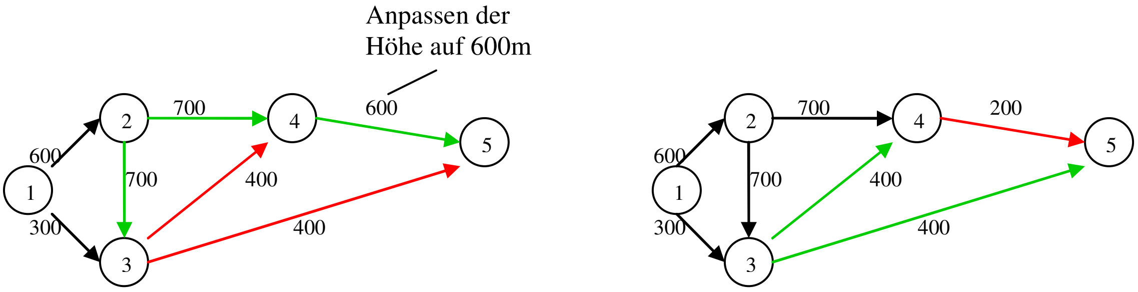


Abbildung 50 Geglättete Landschaft mit Berücksichtigung der Flugrichtung Knoten 2 und Knoten 3

Nachdem der komplette Graph durchlaufen wurde, wird von neuem begonnen, da durch die Änderungen erneut nicht fliegbare Kanten entstanden sein können.

Code:

```
changes=1
while(changes!=0)
    changes=0
    for jeden Knoten
        for jede Ausgehende Kante
            for jede Ausgangskante des Nachfolger
                if richtungsänderung fliegbar
                    if  Steigung>Steigrate
                        Anpassen der Höhe des Knotens
                        changes++
                    if Steigung<Sinkrate
                        Anpassen der Höhe des Nachfolgers
                        changes++
```

Die Komplexität des Algorithmus ist im Best Case linear von der Anzahl der Knoten abhängig, da genau einmal über den gesamten Graphen gegangen und jeder Knoten einmal überprüft wird. Da die Anzahl der Kanten 16 nicht übersteigt, und somit die Anzahl der Kanten konstant ist, kann die Komplexität geschätzt werden auf:

$$O(|V|)$$

Im Worst Case ist die Komplexität n^2, da mindest ein Punkt nach jedem Durchlauf an das Maximum angepasst ist und sich dadurch nicht mehr ändert (siehe 7.3.3). Es kann keine Endlosschleife entstehen, da im Extremfall die Landschaft zu einer glatten Fläche gemacht wird, welche die Höhe des Gelände Maximums besitzt. Dadurch ist gewährleistet, dass eine Lösung nach endlicher Zeit gefunden wird.

Jedoch ist der Algorithmus langsamer als der von 7.3.3, da er eine for-Schleife mehr besitzt und somit um den mittleren Verzweigungsfaktor langsamer ist. Sprich dieser Algorithmus ist im mittel achtmal langsamer als der von 7.3.3. Dieser Algorithmus vereinfacht die Landschaft noch schwächer als der Algorithmus von 7.3.3. Allerdings hat er eine spürbar höhere Laufzeit.

8. Anpassung der Route

Nachdem die günstigste Route auf der geglätteten Landschaft gefunden wurde, werden die Routenpunkte wieder auf die Originalhöhe der Landschaft gesetzt. Die zuvor berechneten Höhen wurden an ein Maximum angepasst. Jedoch muss in den meisten Fällen der Maximumspunkt gar nicht angeflogen werden. Somit könnte man tiefer fliegen als zuvor berechnet und wäre damit noch etwas günstiger. Man nimmt sozusagen den Worst Case bei der Berechnung der geglätteten Landschaft an. Zudem kommt noch, dass Punkte, die in einem Tal liegen, stark angepasst werden. Wenn der Algorithmus aber nun durch ein Tal längs durchroutet, müssen diese Punkte kaum angepasst werden und man kann tiefer fliegen als zuvor berechnet.

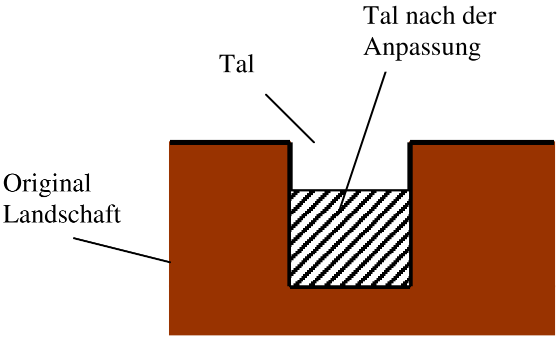


Abbildung 51 Anpassung der Route Tal nach der Anpassung

Die Abbildung 51 veranschaulicht, wie stark ein Tal angepasst wird. Jedoch kann man wesentlich tiefer fliegen, wenn man in Tal-Richtung fliegt. Aus diesen Gründen werden alle Routenpunkte wieder auf die Originalhöhe gesetzt.

Nachdem alle Routenpunkte ihre Originalhöhe wieder besitzen, gibt es Kanten, welche mit den Flugleistungsparametern nicht fliegbar sind. Um dies zu verhindern, wird die Route an die Flugleistungsparameter angepasst. Es besitzt jeder Knoten genau einen Nachfolger. Die Kante zu diesem Nachfolger muss fliegbar sein. Um dies zu gewährleisten, wird vorwärts über die Route gegangen und alle zu starken Gefälle angepasst. Nachdem die komplette Route durchlaufen wurde, wird sie erneut rückwärts durchlaufen und alle zu starken Gefälle (in Durchlaufrichtung) angepasst. Allerdings wird hier das Gefälle mit der negativen Steigrate verglichen.

Nachdem die Route einmal vorwärts und einmal rückwärts durchlaufen wurde und alle Kanten die nicht fliegbar waren, angepasst wurden, ist die Route fliegbar und günstiger als die Route auf Basis der Höhen der geglätteten Landschaft.

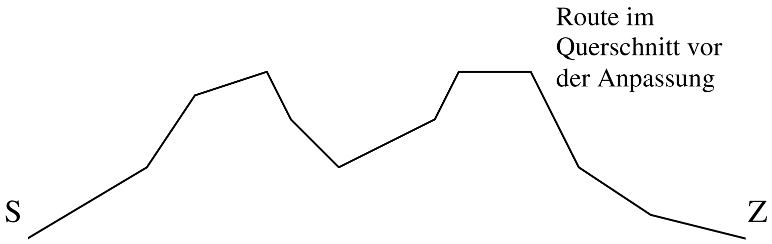


Abbildung 52 Route im Querschnitt vor der Anpassung

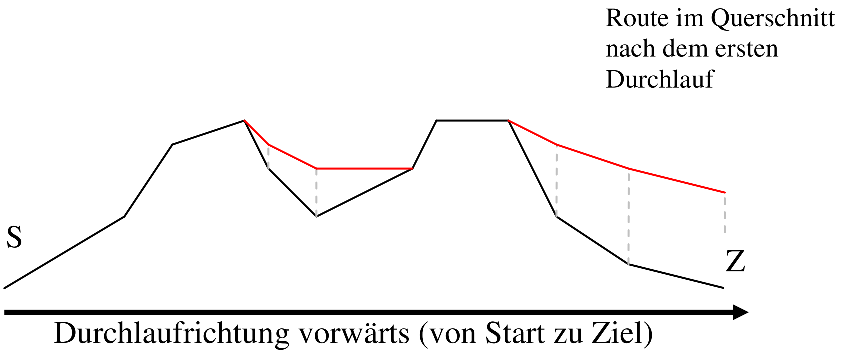


Abbildung 53 Route im Querschnitt nach dem ersten Durchlauf

Im ersten Durchlauf wird die Route vorwärts (von Start zum Ziel) durchlaufen und alle Gefälle überprüft. Die negative Steigung des Gefälles wird mit der Sinkrate verglichen. Sollte die Sinkrate größer sein, so muss das Gefälle angepasst werden. In Abbildung 53 sind die neu angepassten Kanten rot dargestellt.

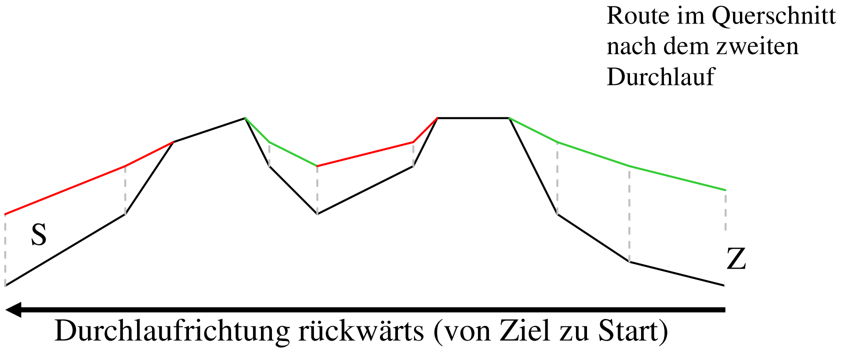


Abbildung 54 Route im Querschnitt nach dem zweiten Durchlauf

Nach dem zweiten Durchlauf wird die Route rückwärts (von Ziel zu Start) durchlaufen und alle Gefälle überprüft. Die Steigung des Gefälles wird mit der negativen Steigrate verglichen. Sollte die negative Steigrate größer sein, so muss das Gefälle angepasst werden. In Abbildung 54 sind die neu angepassten Kanten rot und die zuvor angepassten Kanten grün dargestellt. Nun ist die Route so angepasst, dass sie fliegbar ist.

Code:

```
for jeden Knoten 0 bis n
    if Sinkrate > Steigung zwischen Knoten und Knoten+1
        Passe Höhe an.

for jeden Knoten n bis 0
    if -Steigrate > Steigung zwischen Knoten und Knoten-1
        Passe Höhe an.
```

Die Komplexität ist linear abhängig von der Anzahl der Knoten in der Route. Die Route wird 2-mal durchlaufen und zwar einmal vorwärts und einmal rückwärts. Dadurch ist die Komplexität 2-mal linear von der Anzahl der Knoten in der Route abhängig.

$$O(|KnotenInRoute|) + O(|KnotenInRoute|)$$

$$2 * O(|KnotenInRoute|)$$

9. Erweiterungsmöglichkeiten

- Die Visibility wird derzeit noch mit fest codiertem Sonnenstand berechnet. Eine weiterführende Arbeit wäre, dass die Visibility anhand der Position auf der Erde, der Uhrzeit und des Datums und des daraus resultierenden Sonnenstandes zu berechnen. Analog könnte man nachts anhand der Mondposition am Himmel und seiner aktuellen Leuchtstärke die Visibility entsprechend der Beleuchtung durch den Mond berechnen. Dadurch ist es möglich eine günstige Route abhängig von der gegeben Tageszeit zu finden. Man könnte durch diese Erweiterung das Bedrohungsszenario noch genauer betrachten.

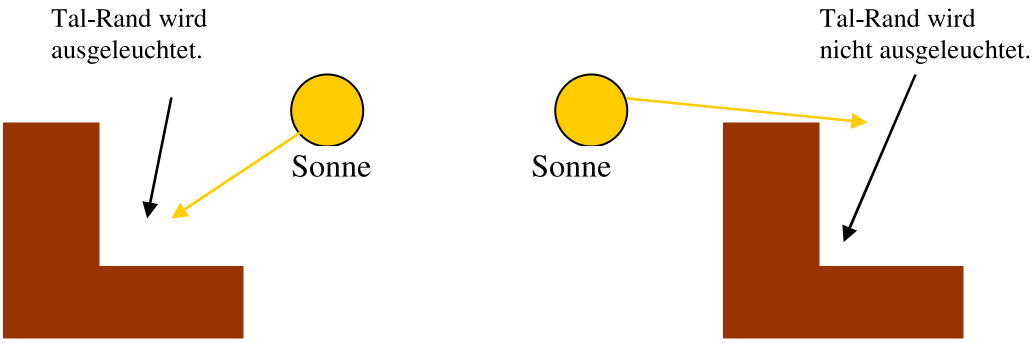


Abbildung 55 Erweiterung Sonnenstand

Die Abbildung 55 verdeutlicht, dass zu verschiedenen Tageszeiten und die dadurch gegebenen Sonnenstände verschiedene Ausleuchtungen entstehen und dadurch Routen günstiger oder teurer werden.

- Eine spezielle Forderung seitens des Auftraggebers ist es, dass man Wetterbergebiete mit unterschiedlichen Sichtweiten definieren kann. Der Minimum Risk Router soll dann für Gebiete mit geringerer Sichtweite die Geschwindigkeit an ein vorgegebenes Maximum anpassen. Durch die Anpassung der Geschwindigkeit würde man andere Steig- und Sinkwinkel und andere Kurvenradien bekommen.

Anhang

Laufzeiten:

Die Laufzeit wurde im Debugmodus im Microsoft Visual Studio gemessen. Das Gelände wurde mit den Standardeinstellungen: Groundspeed 280 Knoten, Steigrate 10 m/s und Sinkrate 10 m/s geglättet. Beim Algorithmus 7.3.3 wird die Visibility für jeden Knoten berechnet, bei Algorithmus 7.3.4 hingegen wird die Visibility für jede Kante berechnet.

Algorithmus 7.3.3

Knotenzahl	Glätten der Landschaft	Berechnung der Visibility	Komplette Laufzeit
2000	0 sec	0 sec	0 sec
8000	0 sec	0 sec	0 sec
60000	1sec	0 sec	2 sec
200000	2 sec	1 sec	8 sec
2000000	67 sec	11 sec	328 sec

Algorithmus 7.3.4

Knotenzahl	Glätten der Landschaft	Berechnung der Visibility	Komplette Laufzeit
2000	0 sec	0 sec	0 sec
8000	1 sec	0 sec	1 sec
60000	8 sec	3 sec	13 sec
200000	57 sec	11 sec	75 sec
2000000	1428 sec	124 sec	2436 sec

Literaturverzeichnis

[1] Dr. Jungnickel, Dieter: „Graphen, Netzwerke und Algorithmen“, Bibliographisches Institut, Zürich 1987

[2] Th. H. Cormen, C. E. Leiserson, R.Rivest, C. Stein: “Algorithmen – Eine Einführung” , Oldenbourg Wissenschaftsverlag GmbH, 2004.

[3] Subat, Volker: „Minimum Risk Router“, Diplomarbeit – Hochschule Zittau/Görlitz (FH), 2004

[5] Guerber, Virginie: „Route Planning Research Project” , Projekt TMLLF EADS Deutschland GmbH Friedrichshafen, 2004

[6] Sedgewick, Robert: „Algorithms“, Addison-Wesley, U.S. 1984

[7] Subat, Volker: „Minimum Risk Router“, Praxissemesterarbeit – Hochschule Zittau/Görlitz (FH), 2003

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Weingarten, 28.02.2007	
Prüfungsort, Abgabedatum	Unterschrift (Vor- und Zuname)