Aufgabe 1: Lisa rennt

Teilnahme-Id: 50997

Bearbeiter/-in dieser Aufgabe:
Jan Niklas Groeneveld

27. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Für Lisas Weg zum Bus werden zuerst zwei Postulate formuliert, die auf der für dieses Problem gültigen Dreiecksungleichung beruhen:

1. Wenn keine Hindernisse im Weg sind, läuft Lisa immer genau den Weg, bei dem sie zum spätest möglichen Zeitpunkt starten kann.

2. Wenn Lisa ihre Bewegungsrichtung ändern muss, geschieht dies ausschließlich an den Ecken der als Polygone dargestellten Hindernisse.

1.1 Der ungestörte Weg

Zuerst muss geklärt werden, wie Lisa laufen muss, wenn ihr Weg frei ist.[1] Dabei bewegt sie sich mit einem zeitlich konstanten Bewegungsvektor $\vec{v_L}$, dessen Betrag als v_L geschrieben wird, während der Busbewegung der ebenfalls zeitlich konstante Vektor $\vec{v_B}$ zugeordnet wird. Der Vektor des Ankunftsortes wird als $\vec{T}$ bezeichnet. Für diesen gilt:

$$\vec{T} = t \cdot \vec{v_B} \tag{1}$$
$$\vec{T} = (t - t_s) \cdot \vec{v_L} + \vec{H} \tag{2}$$

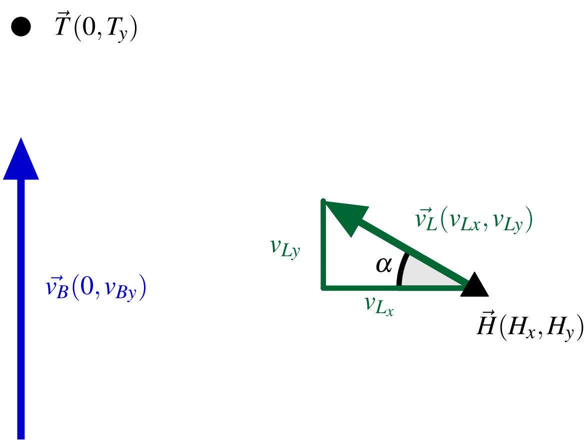

Abbildung 1: Ungestörter Weg von Lisas Haus ($\vec{H}$) zum Treffpunkt mit dem Bus ($\vec{T}$)

[1] Vor den Beispielen der BWINF-Seite ist ein eigenes Beispiel ohne Weg eingefügt.

Dabei ist t die Zeit des Gesamtsystems, dessen Nullpunkt für den Moment definiert ist, in dem der Bus den Ursprung des Koordinatensystems passiert. Die Zeit, zu der Lisa starten muss, wird als t_s bezeichnet. $\vec{H}$ beschreibt die Lage des Punktes, von dem Lisa startet. Im nächsten Schritt wird dieses Gleichungssystem in Komponentenform geschrieben, wobei die Gleichung, die eine Busbewegung in x-Richtung beschreibt, entfällt, weil sich der Bus ausschließlich entlang der y-Achse bewegt. Die Geschwindigkeit Lisas in x-Richtung wird darüber hinaus bereits durch ihre Gesamtgeschwindigkeit und die Bewegung in y-Richtung ausgedrückt ($v_{Lx} = \sqrt{{v_L}^2 - {v_{Ly}}^2}$), und ihr Ankunftsort in x-Richtung liegt auf der y-Achse:

$$T_y = t \cdot v_{By} \tag{3}$$

$$T_y = (t - t_s) \cdot v_{Ly} + H_y \tag{4}$$

$$0 = (t - t_s) \cdot \sqrt{{v_L}^2 - {v_{Ly}}^2} + H_x \tag{5}$$

Durch Lösen dieses Gleichungssystems nach t_s erhält man eine Gleichung, die neben einiger Konstanten von v_{Ly} abhängig ist. Für sie gilt:

$$t_s(v_{Ly}) = -\frac{\left(H_x \cdot (v_{By} - v_{Ly}) - H_y \cdot \sqrt{{v_L}^2 - {v_{Ly}}^2}\right)}{v_{By} \cdot \sqrt{{v_L}^2 - {v_{Ly}}^2}} \tag{6}$$

Weil der spätmöglichste Zeitpunkt für Lisas Start gefragt ist, muss der höchste Punkt dieser Funktion gefunden werden. Dafür wird sie nach v_{Ly} abgeleitet:

$$t_s'(v_{Ly}) = \frac{\mathrm{d}t_s}{\mathrm{d}v_{Ly}} = \frac{H_x \cdot \left({v_L}^2 - v_B \cdot v_{Ly}\right)}{v_B \cdot \left({v_L}^2 - {v_{Ly}}^2\right)^{\frac{3}{2}}} \tag{7}$$

Nach der Nullsetzung ($t_s' = 0$) und Auflösung ergibt sich für $v_{Ly,m}$, der Geschwindigkeit in y-Richtung, bei der Lisa am spätesten starten kann:

$$v_{Ly,m} = \frac{{v_L}^2}{v_B} \tag{8}$$

Bildet man ein rechtwinkliges Dreieck, bei dem diese Geschwindigkeit in y-Richtung die Gegenkathete zu dem Winkel, der am Startpunkt liegt, abbildet, während die Hypotenuse der Gesamtgeschwindigkeit entspricht, muss man, um über den Arkussinus diesen Winkel zu bestimmen, zunächst das Verhältnis bilden, indem man durch die Gesamtgeschwindigkeit teilt:

$$\alpha = \arcsin\left(\frac{{v_L}^2}{v_B \cdot v_L}\right) = \arcsin\left(\frac{v_L}{v_B}\right) \tag{9}$$

Weil das Geschwindigkeitsverhältnis in der Aufgabe 1/2 beträgt, gilt für den Winkel:

$$\alpha = \arcsin\left(\frac{1}{2}\right) = 30^\circ \tag{10}$$

Dieser Winkel von 30° ist folglich unabhängig vom Startpunkt der Winkel, der den spätmöglichsten Startzeitpunkt erlaubt. Lisa wird also – falls möglich – immer unter diesem Winkel zur Straße laufen.

1.2 Der allgemeine Algorithmus

Der eigentliche Algorithmus, der das Problem allgemein löst, besteht aus drei Teilen:

1. Zuerst wird geprüft, ob Lisa direkt von ihrem Haus im 30°-Winkel zur Straße gehen kann. Falls das möglich ist, kann das Programm bereits nach Ausgabe des Startzeitpunktes beendet werden. Wenn das nicht der Fall ist, wird sichergestellt, dass alle Polygone gegen den Uhrzeigersinn notiert sind. Die Gültigkeit der Daten wird dabei vorausgesetzt, die darin besteht, dass keine Polygonecken übereinander liegen.

2. Dann wird ermittelt, welche Wege möglich sind, ohne Hindernisse zu schneiden. Dabei werden die Wege vom Haus zu den Ecken der Polygone, die Wege von den Ecken zur Straße und die Wege zwischen den Polygonecken analysiert und entweder die Weglänge, oder die Information, dass dieser Weg unmöglich ist, gespeichert.

3. Anschließend wird auf Grundlage dieser Informationen der Weg entlang der Polygonecken gesucht, der den spätmöglichsten Startzeitpunkt erlaubt.

1.3 Von einem Punkt aus erreichbare Punkte

Nach dem Einlesen der Werte wird mithilfe eines zweischrittigen Verfahrens bestimmt, ob ein Weg zwischen zwei Punkten möglich ist, wobei zuerst über Winkelberechnungen diejenigen Verbindungen ausgeschlossen werden, die durch das Innere der Polygone des Start- oder Endpunktes gingen. Anschließend werden über die Methoden der analytischen Geometrie die Verbindungen ausgeschlossen, die andere Polygone schnitten. Für die eindeutige Definition der Polygone ist es unabdingbar, dass die Punkte der Polygone entweder im Uhrzeigersinn oder gegen den Uhrzeigersinn angegeben sind. Die Kanten der Polygone werden dabei als Vektoren angegeben. Die zu prüfende Strecke zwischen zwei Punkten $\vec{A}$ und $\vec{B}$ wird mit einer Geradenbeschreibung beschrieben. Für den Richtungsvektor $\vec{w}$ gilt: $\vec{w} = \vec{b} - \vec{a}$. Mithilfe des Skalars (r) wird dann die eine Geradengleichung bestimmt: $\vec{x} = \vec{a} + r \cdot \vec{w}$. Alle $\vec{x}$, für die gilt: $0 \leq r \leq 1$, liegen auf der Strecke zwischen $\vec{A}$ und $\vec{B}$. Abbildung 2 stellt eine beispielhafte Situation dar.

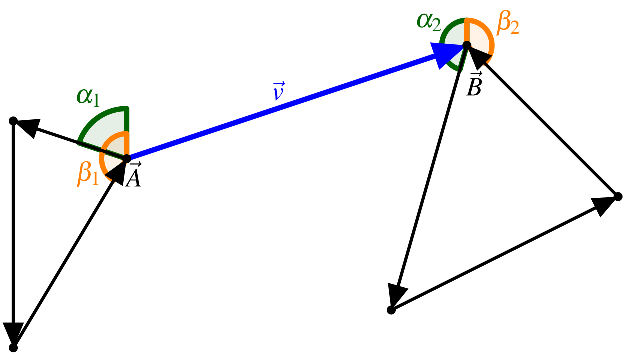

Abbildung 2: Darstellung einer Verbindung $\vec{v}$ zwischen zwei Punkten $\vec{A}$ und $\vec{B}$

Während es für den späteren Teil dieser Prüfung unbedeutend ist, ob die Notation eines Polygons im Uhrzeigersinn oder gegen den Uhrzeigersinn stattfindet, ist dies für den Teil der Winkelberechnungen von entscheidender Bedeutung, wie im Anschluss erläutert wird. Für alle Eckpunkte existiert ein Winkelbereich, innerhalb dessen keine Verbindungen zulässig sind. Dieser wird durch zwei Winkelangaben eindeutig definiert: Der Winkel, der die obere Grenze des verbotenen Winkelbereiches festlegt (hier immer als α bezeichnet), und den, der die untere Grenze festlegt (hier immer als β bezeichnet). Die Winkelrose wird dabei so definiert, dass null-Grad exakt nach oben zeigt, links Winkel, die einen Wert kleiner als null besitzen, liegen, und rechts die Winkel mit einem Wert größer als null. Es sind also Winkel von über -180° bis 180° möglich.[2] Zu diesem Zeitpunkt der Programmlaufzeit sind bereits alle Polygone gegen den Uhrzeigersinn notiert, weswegen die folgende Beschreibung auch auf dieser Grundlage erfolgt.[3] Die Winkel α, die den oberen Rand der jeweils verbotenen Bereiche definieren, werden bestimmt, indem der Winkel des Vektors berechnet wird, der von dem betrachteten Punkt zum folgenden führt. Die Winkel β definieren den unteren Rand des verbotenen Bereiches und werden als Winkel des Vektors des jeweils vorherigen Punktes zum aktuellen berechnet. Wenn man jetzt prüfen möchte, ob der Winkel γ, unter dem der Vektor der Verbindung ($\vec{v}$) steht, durch das Innere des Polygons des Startpunktes führt, gelten zwei Regeln für unterschiedliche Anordnungen von α und β. Um diese Regeln für den Zielpunkt gleichermaßen anwenden zu können, wird für γ der Winkel genommen, unter dem die Umkehrung von $\vec{v}$ steht. Unter Verweis auf das linke Polygon der Abbildung 2 gilt folgende erste Regel: Wenn γ kleiner ist als α_1 und größer ist als β_1, ist die Verbindung verboten. Das gilt, falls der den oberen

2 Zeigt ein Vektor direkt nach unten, wird er als 180° beschrieben, weshalb nur Winkel von mehr als -180° auftreten.

3 Um Polygone, die im Uhrzeigersinn notiert sind, umzudrehen, wird für jedes Polygon der Punkt mit der geringsten y-Koordinate gesucht. Dann werden die im Text erklärten Winkel α und β miteinander verglichen, und das Polygon wird dann als im Uhrzeigersinn notiert identifiziert, wenn α kleiner ist als β. Dann wird die Notationsreihenfolge umgekehrt und die Strecken und die Winkel für dieses Polygon erneut berechnet.

Rand definierende Winkel α_1 einen größeren Wert hat als der den unteren Rand definierende Winkel β_1. Bei dem Fall des rechten Polygons sieht man, dass der Übergang von $180° \to -180°$ innerhalb des verbotenen Bereiches liegt. Wenn man für einen solchen Fall prüfen möchte, ist offensichtlich, dass die erste genannte Regel unzweckmäßig wäre. Wenn der den unteren Rand definierende Winkel β_2 also einen größeren Wert aufweist als der den oberen Rand definierende Winkel α_2, ist eine Verbindung genau dann verboten, wenn γ größerer ist als β_2 oder kleiner ist als α_2.

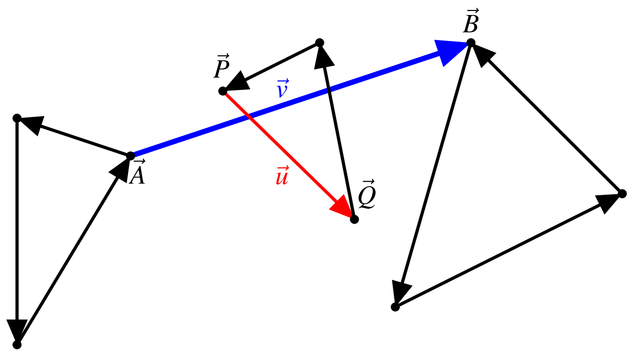

Abbildung 3: Ein Hindernis macht die direkte Strecke von $\vec{A}$ nach $\vec{B}$ unmöglich

Wenn diese Winkelanalyse nicht zum Ausschluss einer Verbindung führt, werden die anderen Strecken zwischen den Polygonecken darauf überprüft, ob sie den direkten Weg zwischen $\vec{A}$ und $\vec{B}$ stören. Dazu wurden bereits zuvor Geradengleichungen für die Strecken zwischen benachbarten Ecken der Polygone aufgestellt, die die Polygonkanten beschreiben. Obwohl diese bereits bei der oberen Prüfung für die Berechnung der Winkel α und β verwendet wurden, wird hier noch einmal näher auf sie eingegangen. Die Polygonkanten wurden beschrieben, indem ein Vektor als Differenz der Vektoren der Eckpunkte, beispielsweise $\vec{P}$ und $\vec{Q}$, bestimmt wurde: $\vec{u} = \vec{Q} - \vec{P}$. Die Gleichung wird dann mithilfe eines Skalars (s) und des Stützvektors ($\vec{P}$) aufgestellt: $\vec{x} = \vec{P} + s \cdot \vec{u}$. Punkte, für die $0 \leq s \leq 1$ gilt, liegen auf der Strecke zwischen $\vec{P}$ und $\vec{Q}$. Um die Lagebeziehung dieser Gerade und der der Verbindung von $\vec{A}$ nach $\vec{B}$ aussagekräftig zu untersuchen, muss zunächst geprüft werden, ob die Richtungsvektoren $\vec{v}$ und $\vec{u}$ Vielfache voneinander sind. Wenn das der Fall ist, wird ein direkter Weg von $\vec{A}$ nach $\vec{B}$ durch die Verbindung von $\vec{P}$ nach $\vec{Q}$ nie gestört, denn selbst wenn die Geraden übereinander liegen, kann dieser Weg gegangen werden. Das ist der Fall, wenn sich Lisa entlang einer Polygonkante bewegt. Sind die Vektoren $\vec{v}$ und $\vec{u}$ keine Vielfachen voneinander, werden die Geradengleichungen gleichgesetzt, um konkrete Werte für r und s zu berechnen, denn in einer zweidimensionalen Ebene wird es in diesem Fall immer einen Schnittpunkt geben: $\vec{A} + r \cdot \vec{v} = \vec{P} + s \cdot \vec{u}$. Anhand folgender Regeln kann nach dem Lösen des Gleichungssystems bestimmt werden, ob die Verbindung möglich ist:

1. Wenn für beide Skalare r und s ein Wert, der größer als null und kleiner als eins ist, ermittelt wird, schneiden sich die Geraden in dem Bereich zwischen den jeweiligen Punkten, was die Verbindung verbietet.

2. Wenn für das Skalar s, das für die Beschreibung der Polygonkante verwendet wird, ein Wert größer null und kleiner eins ermittelt wird, und genau eins oder null für das Skalar r, das für die Beschreibung der Strecke verwendet wird, liegt der Start- oder Endpunkt Lisas potentieller Bewegung auf einer Polygonkante. In diesem Fall ist eine Verbindung verboten.[4]

3. Wenn das Skalar s entweder genau null oder genau eins beträgt, und r kleiner als eins und größer als null ist, schneidet der zu prüfende Weg zwischen zwei Ecken entweder den Startpunkt oder den Endpunkt der geprüften Polygonkante. In diesem Fall muss eine Verbindung nicht prinzipiell verboten sein, aber sie wird dennoch als verboten markiert, um einen ungewollten Effekt zu unterbinden, nämlich die nicht vollständig ausgeschlossene Durchtunnelung eines Hindernisses. Einen schlechteren Weg wird man durch diese Regel jedoch niemals erreichen, da der Punkt selber, welcher in diesem Fall überflogen würde, bei seiner Prüfung als erlaubt markiert werden kann. Von diesem kann dann zu dem ursprünglich angestrebten gegangen werden, sofern keine anderen Hindernisse dies verhindern.

Um diesen Weg von $\vec{A}$ nach $\vec{B}$ vollständig zu verifizieren, muss diese Berechnung für sämtliche Kanten der Polygone durchgeführt werden. Wenn für eine Polygonkante ermittelt wird, dass sie die Verbindung von $\vec{A}$ nach $\vec{B}$ stört, kann sofort abgebrochen werden, und mit der nächsten potentiellen Verbindung fortgefahren werden. Eine solche Analyse wird für alle Verbindungen zwischen den Polygonecken untereinander, zwischen Lisas Haus und den Polygonecken und zwischen den Polygonecken und der Straße unternommen, wobei der Weg von Punkt zur Straße immer unter dem 30° Winkel liegt, sodass den y-Wert des Straßenschnittpunktes gilt: $A_y = P_y + P_x \cdot \tan(30°)$. Dabei ist P_x die x-Koordinate des Punktes und P_y die y-Koordinate.

1.4 Laufzeit des Sichtbarkeits-Algorithmus

Um nach dem oben beschriebenen Verfahren zu ermitteln, welche Wege gegangen werden können, müssen drei Prozesse verschachtelt ausgeführt werden: Von jeder Ecke muss im Worst-Case der Weg zu allen anderen Ecken geprüft werden, also $n-1$ (der Worst-Case liegt nicht vor, wenn bereits Ecken über die Winkel ausgeschlossen werden konnten).

[4] Diese Regel wurde auf dem BWINF-Frageforum auf Nachfrage eines Teilnehmenden genannt. Unter folgender Adresse lässt sich die Fragestellung nachlesen: *https://www.einstieg-informatik.de/community/forums/topic/625/fragen-zur-2-runde-aufgabe-1-lisa-rennt/view/page/1*

Um einen Weg zu verifizieren, muss nachgewiesen werden, dass auf diesem Weg keine Überschneidung mit Kanten vorliegt. Zu n Ecken gehören auch n Kanten, sodass auch hier im Worst-Case n Prüfungen notwendig sind (weniger Prüfungen sind notwendig, wenn bereits früher eine Überschneidung festgestellt wurde und daher früher abgebrochen werden kann). Um keine der Ecken auszulassen, muss dieses Verfahren wiederum für n Ecken ausgeführt werden. Aus diesen Überlegungen folgt, dass dieser Teilalgorithmus mit $O(n^3)$ zu klassifizieren ist. Diese Worst-Case Laufzeit kann in einem mittleren Fall mit einem linearen Faktor $k \leq \frac{1}{2}$ multipliziert werden, da ein Teil der Ecken über die Winkel ausgeschlossen wurde, bei Überschneidungen im Durchschnitt nach weniger als n Kanten abgebrochen werden kann. Der entscheidende Fall, weswegen k immer kleiner oder gleich $\frac{1}{2}$ ist, ist der, dass Wege von einer Ecke zu einer anderen bereits markiert wurden und nicht noch einmal gerechnet werden müssen, wenn der Weg von der anderen zur ersten gefragt wird. Das halbiert die Anzahl der notwendigen Berechnungen. Im Best-Case wird der Algorithmus gar nicht aufgerufen, da dann keine Hindernisse Lisas Weg beeinflussen.

1.5 Algorithmus zur Bestimmung des optimalen Weges

Anschließend wird auf Grundlage dieses erst einmal relativ einfachen Bruteforcealgorithmus der Weg bestimmt, bei dem Lisa am spätesten losgehen muss: Von ihrem Startpunkt aus werden alle möglichen Punkte ausprobiert, und jeweils die bekannten Weglängen für diesen Weg addiert. In der nächsten Rechentiefe werden von ihnen ausgehend wieder alle erreichbaren Punkte durchprobiert. Dabei muss sichergestellt sein, dass die bisher auf einem Weg genutzten Ecken weitergegeben und danach nicht wieder angelaufen werden, um Schleifen zu verhindern. Wenn dabei ein Punkt erreicht wird, von dem ein direkter Weg zur Straße unter dem besagten 30°-Winkel möglich ist, kann nach Addition der letzten Wegstrecke die Zeit t_s bestimmt werden, zu der Lisa spätestens losgehen muss. Für sie gilt:

$$t_s = \frac{T_y}{v_B} - \frac{s_L}{v_L} \tag{11}$$

Dabei ist s_L der Betrag der Strecke, die Lisa auf diesem Weg zurücklegen muss. Ist t_s größer als die Zeit, die vorher als der spätest mögliche Zeitpunkt zum Losgehen gespeichert war, wird der vorherige Weg durch den aktuell berechneten ersetzt. Durch dieses Vorgehen stellt man zwar immer sicher, dass der optimale Weg erreicht wird, allerdings ist die geringe Effizienz des bisherigen Algorithmus nicht rechtfertigbar. Um diesen Teil des Algorithmus drastisch zu optimieren, wurde er um drei Zusätze erweitert:

1. In jedem Punkt wird geprüft, ob die späteste bisher bekannte Startzeit selbst dann nicht mehr überschritten werden kann, wenn von diesem Punkt direkt zur Straße

gegangen werden könnte. In diesem Fall ist jedes weitere Durchprobieren zwecklos, da keine bessere Zeit mehr erreicht wird.

2. Die nächste Optimierung macht sich zunutze, dass viele Wege bereits theoretisch ausgeschlossen werden können. So existieren einige Möglichkeiten, um von einem beliebigen Punkt aus andere Punkte zu erreichen. Wird davon ein Punkt für den Weg ausgewählt, kann es sein, dass von diesem Punkt aus zahlreiche der vorher erreichbaren Punkte weiterhin erreichbar sind und als nächste Stufen ausprobiert werden. Das ist jedoch unerwünscht, weil diese Punkte, wenn sie ausprobiert werden sollen, bereits von der vorherigen Stufe aufgerufen werden. Alles andere wäre ein Umweg und auf jeden Fall aufwändiger. Also werden alle Punkte, die schon einmal erreichbar waren, für den folgenden Weg ausgeschlossen.

3. Weil es immer noch vorkommen kann, dass eine Ecke über unterschiedliche Wege erreicht wird, bietet es sich an, ein weiteres Abbruchkriterium zu definieren: Für jeden Punkt wird die minimale Strecke gespeichert, nach der der Punkt bisher erreicht wurde. Wenn der Punkt nun erneut erreicht wird, wird geprüft, ob die bisher zurückgelegte Strecke länger ist als die bisher geringste. In diesem Fall wird sofort abgebrochen, da man durch weiteres Durchrechnen keine weiteren Informationen gewinnen kann. Ist der bisher zurückgelegte Weg hingegen kürzer, wird er als geringster bisher bekannter Weg für diesen Punkt gespeichert.

1.6 Laufzeit des Algorithmus zur Wegermittlung

Der Vorteil dieser drei Optimierungen besteht darin, dass trotz eines riesigen Effizienzgewinnes immer noch in jedem Fall der beste mögliche Weg gefunden wird, da kein Weg abgebrochen wird, wenn er prinzipiell noch eine bessere Lösung liefern könnte. Die Laufzeit dieses Algorithmus genau zu determinieren, erweist sich wegen der zahlreichen Optimierungen als äußerst schwierig. Der Worst-Case $O(e^n)$, der den Algorithmus ohne Optimierung beschreibt, ist für das optimierte Verfahren in keinster Weise aussagekräftig, da mit Optimierungen nur noch ein sehr geringer Anteil der Möglichkeiten durchgerechnet werden müssen. Neben der Anzahl der Ecken ist die Laufzeit des Verfahrens in hohem Maße davon abhängig, wie viele Verbindungen von einer Ecke zu einer anderen bestehen, also wie die Hindernisse in Lisas Umgebung angeordnet sind. Je weniger mögliche Verbindungen von einer Ecke zu anderen existieren, desto geringer die Laufzeit. Zur Laufzeit in Abhängigkeit der Anzahl der Polygonecken lässt sich aus den Erfahrungen mit den durchgerechneten Beispielen in grober Näherung lineares Wachstum ermitteln, wie das Diagramm in Abbildung 4 zeigt. In diesem Diagramm wurde die Anzahl der vom Wegsuchalgorithmus durchgerechneten Knoten auf die Anzahl der Polygomecken aufgetragen. Daraus wird auch ersichtlich, dass einzelne Punkte stark abweichen können. In jedem Fall lässt sich sagen, dass die Laufzeit dieses Teilalgorithmus signifikant geringer ist als der zur Erstellung des Sichtbarkeitsgraphen, der (unter Berücksichtigung eines linearen Faktors) proportional zu n^3 wächst.

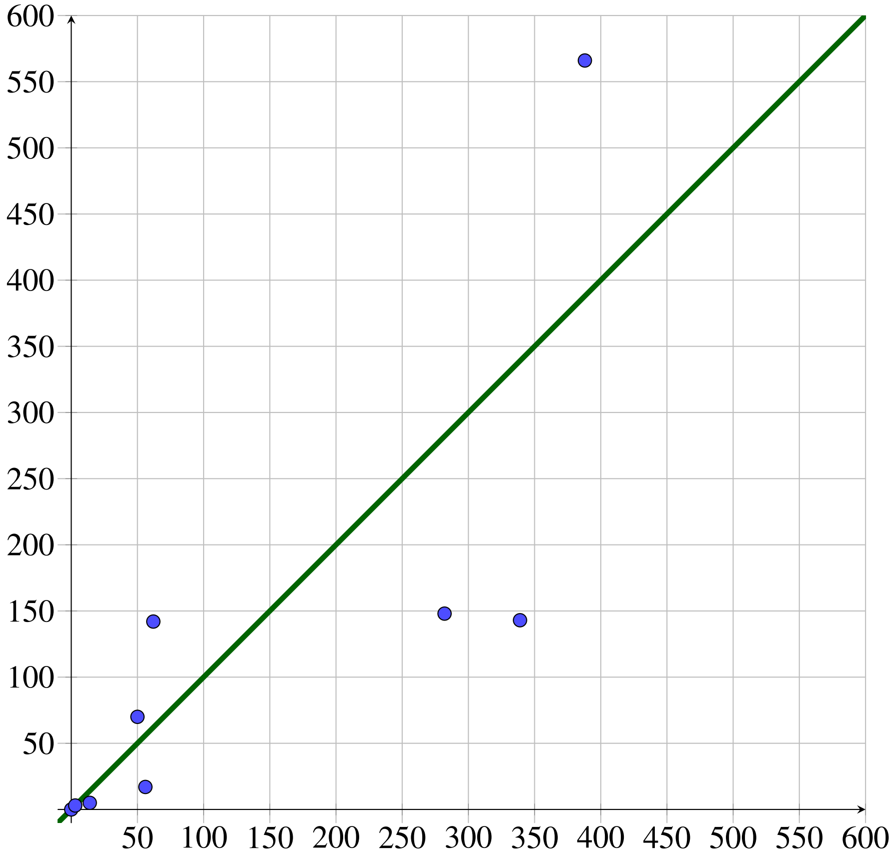

Abbildung 4: Auswertung der Beispieldaten

2 Umsetzung

Die C-Implementierung des Problems ist in Abbildung 5 überblicksartig dargestellt. Dabei sind auf gelben Hintergrund die Namen wichtiger globaler Variablen dargestellt, die von zahlreichen Funktionen genutzt werden, und deren Datentypen im Text erklärt werden. Die Funktionen sind in ihrer Aufrufreihenfolge aufsteigend angeordnet, wobei ein Pfeil immer einen Aufruf von einer anderen Funktion ausgehend darstellt. Funktionen, die nicht grün gefärbt wurden, haben besonderen Einfluss auf das Programm, wie im Fall von `lisadirekt()`, weil das Programm nach dieser Funktion sofort beendet werden kann, oder bedienen weitere im Text erläuterter Konzepte. Pfeile auf die Variablen bedeutet, dass von der ausgehenden Funktion schreibender Zugriff stattfindet. Um die Übersicht zu

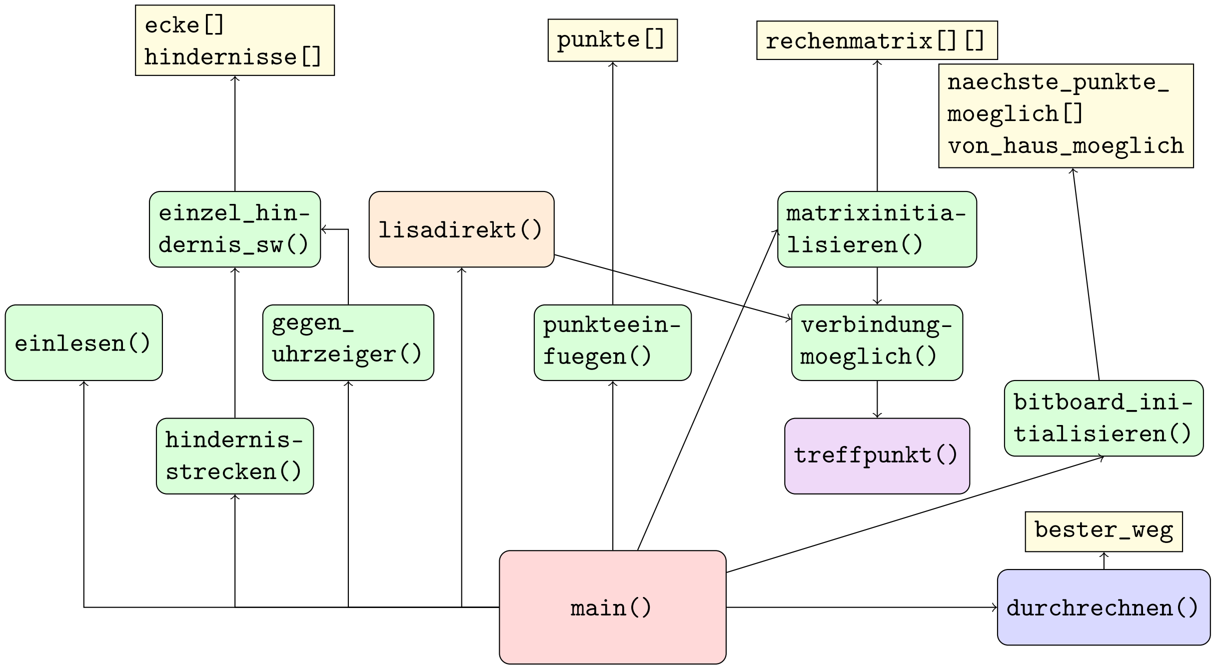

Abbildung 5: Übersicht die Anordnung einzelner Funktionen und wichtiger globaler Variablen

wahren, wurden nicht alle Funktionen in der Abbildung dargestellt, sondern werden teils auch im Text an passender Stelle erwähnt. Wie der Abbildung zu entnehmen ist, beginnt das Programm mit dem Einlesen der Textdatei, in der die Angaben über die Ecken der Hindernisse gespeichert sind, und speichert sie in einem Array der Struktur `Hindernis` (`hindernisse[]`), die wiederum die Anzahl der Ecken eines Polygons und einen Array der Struktur `Punkt` (`punkte[]`) aufweist. Letztere speichert die x- und y- Koordinaten eines Punktes als Ganzzahlen[5]. Darüber hinaus sieht die Struktur `Hindernis` einen Array vom Typ `Gerade` vor. Hier werden die Beschreibungen der Kanten des Polygons gespeichert, von denen es genauso viele gibt wie Ecken. Die Funktion `hindernisstrecken()` iteriert in einer Zählschleife durch die Anzahl der Hindernisse und ruft für jedes Hindernis die Funktion `einzel_hindernis_sw()` auf, die wiederum durch die Ecken pro Hindernis iteriert, um über die Funktion `geradebestimmen()` Geradenbeschreibungen des Typs `Gerade` für die Kanten zu ermitteln. Dabei ist der Geradenbeschreibung mit der höchsten Indexzahl immer die Kante zwischen der Ecke mit der höchsten Indexzahl und der ersten gespeicherten Ecke zugeordnet. Die Struktur `Gerade` enthält dabei den Ausgangs-

[5] Die Rundungen auf Ganzzahlen stellen einen vernachlässigbaren Genauigkeitsverlust dar, denn ausschließlich beim letzten Punkt, dem des Straßenschnittes, kann es geringfügige Abweichungen geben. Diese liegen bei den Beispielen bei deutlich weniger als einem Prozent, was in dieser Genauigkeit durch den Sachzusammenhang nicht gefordert wird, und die Zeit am Schluss auf Sekunden gerundet ausgegeben wird.

punkt und den Richtungsvektor, der als Differenz aus den zwei Eckpunkten berechnet wird, wie das im Abschnitt »Grundidee« erläutert wurde. Darüber hinaus werden für jede dieser Ecken die Winkel gespeichert, die den für Bewegungen verbotenen Innenbereich der Polygone begrenzen. Dazu wird eine angepasste Arkustangensfunktion verwendet.

Daraufhin wird aus der `main()`-Methode die Funktion `gegen_uhrzeiger()` aufgerufen, um das Dreieck wie oben gefordert gegen den Uhrzeiger zu drehen. Sie bestimmt den Eckpunkt mit dem niedrigsten y-Wert für jedes Polygon und vergleicht die Winkel wie beschrieben. Sollte das Polygon im Uhrzeigersinn notiert worden sein, wird die Reihenfolge der Punkte vertauscht, und die Funktion `einzel_hindernis_sw()` für dieses Polygon erneut aufgerufen.

Im Anschluss daran wird die Funktion `lisadirekt()` aufgerufen. Sie prüft über folgendes Verfahren, ob Lisa von ihrem Haus direkt zur Straße gelangen kann: Über die Funktion `strasseschnitt()` wird der Punkt ermittelt, an dem Lisa die Straße erreichen würde. Der Funktion `verbindungmoeglich()` werden dann die beiden Punkte übergeben, die, wenn eine Verbindung möglich ist, die Länge des Weges, und andernfalls den auf -1 definierten Wert `IMPOSSIBLE` zurückgibt. Dazu wird die bereits genannte Funktion `geradebestimmen()` verwendet, der der Ankunftspunkt und der Punkt mit den Koordinaten des Hauses übergeben werden. Dann wird durch den Array der Hindernisse und durch den Array der Kantenbeschreibungen der Hindernisse iteriert, um mit dieser hier enthaltenen Geradenbeschreibung und derjenigen, die den Weg zwischen Haus und Straße beschreibt, die Funktion `treffpunkt()` aufzurufen.[6] Sie lässt zuerst die Funktion `vielfache()` prüfen, ob die Richtungsvektoren Vielfache voneinander sind. In diesem Fall wird `FALSE` zurückgegeben, um deutlich zu machen, dass kein kritischer Schnittpunkt vorliegt. Andernfalls wird ein geeignetes Verfahren zur Lösung der Schnittgleichung angesetzt. Wenn mindestens einer der vier Richtungskomponenten[7] gleich null ist, lässt sich das Lösen der Gleichung so vereinfachen, dass feste Lösungsformeln implementiert sind, ansonsten wird eine 3x2-Matrix angelegt, mit der das Gleichungssystem über das Gauß-Verfahren gelöst wird. Dieser Teil ist im Quellcode hinreichend kommentiert und muss hier nicht ausführlich erläutert werden. Um einen Schnittpunkt innerhalb des gefragten Bereiches zu haben, werden die im Abschnitt »Grundidee« erläuterten Regeln angewandt. Liegt ein solcher Schnittpunkt vor, durch den die Verbindung verboten ist, wird die Funktion `treffpunkt()` `TRUE` zurückgeben. Sollte die Funktion `verbindungmoeglich()` dieses Ergebnis erhalten, wird sofort abgebrochen und über den Wahrheitswert `FALSE`

[6] In diesem Fall wird die Funktion `treffpunkt()` niemals wegen der Winkelbedingungen, die am Anfang abgefragt werden, eine Unmöglichkeit der Verbindung feststellen, da weder das Haus als Startpunkt, noch die Straße als Ziel für bestimmte Winkel verbotene Bereiche aufweisen. Das ist als Eigenschaft der Punkte in der Struktur entsprechend markiert.

[7] x- und y-Richtung Lisas potentieller Bewegung und x- und y- Richtung des Kantenvektors

zurückgegeben, dass die Verbindung nicht möglich ist.

Anschließend iteriert die Funktion `punkteeinfuegen()` wieder durch den Array der Hindernisse und den darin enthaltenen Array der Eckpunkte, um jeden Eckpunkt in den Array `punkte`, der Variablen der Struktur `Punkt` sammelt, einzutragen und die Gesamtzahl an Ecken in der Variable `ges_punkte` zu speichern. Dies dient dem Zwecke einer eindeutigen Zuordnung einer Ecke zu einer Indexzahl.

Die Funktion `matrixinitialisieren()` iteriert zuerst durch den Array `punkte`, um den Array, der Variablen der Struktur `Ankunft` speichert, zu füllen. Dieser dient dazu, die Verbindung von einem Eckpunkt zum Schnittpunkt mit der Straße zu beschreiben, und speichert als Wahrheitswert, ob eine Verbindung möglich ist. Darüber hinaus wird die y-Höhe des Schnittpunktes als Ganzzahl und die Entfernung von dem Punkt zum Straßenschnittpunkt gespeichert. Nach Ermittlung dieser Werte über die bereits erläuterten Funktionen `strasseschnitt()` und `verbindungmoeglich()`[8] werden diese hier eingetragen. Zu erwähnen ist hier noch, dass die Entfernung zur Straße selbst dann noch eingetragen wird, wenn keine direkte Verbindung möglich ist, damit sie für die erste Optimierung zur Verfügung steht. Im Anschluss werden die Entfernungen, bzw. die Unmöglichkeit der Verbindungen vom Haus zu den Eckpunkten in den Dezimalzahl-Array `haus_zu_punkt` eingetragen und im letzten Schritt dieser Funktion wird mithilfe einer verschachtelten Zählschleife durch den Array `punkte[]` iteriert, um die Dezimalzahl-Matrix `rechenmatrix[][]` auszufüllen. Hier werden die Entfernungen, bzw. die Unmöglichkeit der Verbindungen der Polygonecken untereinander gespeichert, wobei beachtet wird, dass die Matrix eine Spiegelachse, die von der nullten Zelle in x- und y-Richtung entlang der Zellen führt, bei denen sich x- und y- Wert gleichen. Dadurch ist nur die Hälfte der Schnittanalysen notwendig, die hier zwischen den Punkten nach bekanntem Muster ausgeführt wird. Der Schleifendurchlauf, bei dem sich die Werte für die Zelle in x- und y-Richtung gleichen, wird dadurch übersprungen, dass die zweite Zählvariable so beginnt, dass innerhalb der Matrix nur ein Dreieck gerechnet wird.

Die Funktion `durchrechnen()`, die die eigentliche Wegsuche übernimmt, bekommt als Übergabeparameter eine Struktur vom Typ `Wegbeschreibung`, die folgende Informationen über einen vollständigen Weg speichert: Einen Array mit den Eckpunkten, die Lisa berührt, den spätesten Zeitpunkt, zu dem Lisa losgehen kann, und die Länge des zurückgelegten Weges, die für die Zeitberechnung notwendig ist. Darüber hinaus werden die Informationen über die bereits ausgeschlossen Punkte übergeben, deren genaue Kodierung im folgenden Absatz genauer erläutert wird.[9] Der grundsätzliche Aufbau dieser

[8] Ab jetzt wird die Winkelprüfung entscheidend sein.

[9] Darüber hinaus werden der Index, unter dem der aktuelle Punkt in allen hierfür wichtigen Arrays und der Matrix gefunden werden kann, und die Tiefe der Rekursion übergeben, da die Wegsuche rekursiv aufgebaut ist.

Funktion ist sehr einfach: Nachdem der aktuelle Punkt in die Wegbeschreibung mit der Rechentiefe als Index eingetragen wurde, wird geprüft, ob von dem aktuell betrachteten Punkt eine direkte Verbindung zur Straße möglich ist. Wenn das der Fall ist, muss für das Abspeichern dieses Weges als bester Weg sichergestellt sein, dass die Startzeit diejenige aus der vorherigen Wegbeschreibung überschreitet, weswegen eine Abfrage eingebaut wird. Wenn kein direkter Weg möglich ist, iteriert eine Zählschleife durch die Anzahl der anderen Eckpunkte und ruft, falls ein Punkt als möglich angegeben ist, die Funktion `durchrechnen()` rekursiv auf. Dabei wird die aktuelle Wegbeschreibung, die ausgeschlossenen Punkte, die bisherige Strecke plus diejenige, die zum Erreichen des nächsten Punktes notwendig ist, der Index des nächsten Punktes und die mit eins addierte Rechentiefe übergeben. Die ausgeschlossenen Punkte sind erst einmal dafür zuständig, dass das erneute Aufrufen eines bereits aufgerufenen Punktes und der damit verbundenen Endlosschleife verhindert wird, was in der zweiten Optimierung dadurch ergänzt wird, dass hier weitere auszuschließende Punkte notiert werden. Die erste und zweite Optimierung werden als Abfragen an den Anfang der Funktion geschrieben. Erstere wird dadurch verwirklicht, dass die Startzeit bereits hier einmal unter der Annahme berechnet wird, dass eine direkte Verbindung zur Straße möglich ist. Unterschreitet diese Startzeit bereits die beste bisher bekannte, wird die Funktion hier über die Anweisung `return` abgebrochen[10]. Die dritte Optimierung wird direkt danach implementiert, indem eine Abfrage auf den Dezimalzahlen-Array `min_strecke_zu_punkt` mit dem Index des aktuellen Punktes zugreift, um zu prüfen, ob die bisher zu diesem Punkt benötigte Strecke geringer ist als die geringste bekannte Strecke zu diesem Punkt. Nur in diesem Fall ergibt ein Weiterrechnen Sinn, und die bisher benötigte Strecke wird als geringste bisher bekannte benötigte Strecke in den Array eingetragen.

Die Information, welche Punkte bereits ausgeschlossen werden, wird binär in einem Array unsignierter 64-Bit-Integer gespeichert. Ein Array wird deswegen verwendet, weil mehr als 64 Eckpunkte vorkommen können. Auf die Information zu einem bestimmten Punkt wird zugegriffen, indem das Element des Arrays als Ergebnis einer Division durch 64 definiert wird, und die Nummer des Bits in diesem 64-Bit-Integer ergibt sich durch eine Modulooperation des Index durch 64. Diese Bitmaps werden je nach Anwendungszweck innerhalb dieses Programms unterschiedlich verwendet. In dem `Bitmap`-Array `naechste_punkte_moeglich[]` werden die Punkte, die von einem Punkt des entsprechenden Index erreichbar sind, mit eins markiert, die unerreichbaren als null. Das gleiche Prinzip wird bei der Bitmap `von_haus_moeglich` verwendet, der die Möglichkeiten, zu welchen Ecken Lisa starten kann, speichert. Weil gemäß des im Abschnitt »Grundidee« erläuterten Prinzips, dass Punkte, die auf einem Weg des Suchbaums bereits einmal er-

[10] Um geringe Unsicherheiten zu vermeiden, wird hier ein sehr geringer Toleranzbereich eingebaut

reichbar waren, fortan nicht ausprobiert werden sollen, sind alle Punkte, die Lisa von ihrem Haus erreichen konnte, nach der nullten Rechentiefe ausgeschlossen. Das bedeutet, dass bei Aufruf der Funktion `durchrechnen()` zur nullten Rechentiefe, bei dem alle erreichbaren Punkte einmal als Index übergeben wurden, diese Bitmap der möglichen Punkte als Speicher für die ausgeschlossen Punkte übergeben wird. Weil bei dieser Übergabe der Punkt selber durch die Bitmap schon ausgeschlossen ist, kann das Problem der Endlosschleife bequem gelöst werden. Innerhalb der Funktion wird die Bitmap der ausgeschlossen Punkte, in der ausgeschlossene als eins und nicht ausgeschlossene als null markiert sind, für zwei Aspekte benötigt. Einerseits lassen sich die Punkte, die in der nächsten Rekursionsstufe ausprobiert werden sollen, effizient dadurch bestimmen, dass die Bitmap der ausgeschlossen Punkte umgekehrt wird, also Nullen zu Einsen und Einsen zu Nullen werden, und über einen bitweisen AND-Operator mit der Bitmap der Möglichkeiten des aktuellen Punktes verknüpft werden. Im Ergebnis dieser Operation sind auszuprobierende Punkte als Einsen vermerkt. Andererseits lassen sich die Punkte, die ab der nächste Rekursionsstufe ausgeschlossen sind, ebenfalls effizient durch eine bitweise OR-Verknüpfung der bereits ausgeschlossen Punkte und der Möglichkeiten des aktuellen Punktes bestimmen.

3 Erweiterungsausblick: Lisa als Kajakfahrerin

Eine sehr interessante Erweiterung der Problemstellung bestünde darin, die Situation in arktische Gegenden zu verlegen. Lisa hat dann weiterhin ihren Wohnsitz an einem bestimmten Punkt in der Ebene, muss aber mit einem Schiff zur Schule gebracht werden. Weil sie den Termin der Schiffsabfahrt nicht rechtzeitig erreicht, versucht sie mit ihrem Kajak das Schiff zu erreichen, das sich mit einer bestimmten Geschwindigkeit entlang der y-Achse bewegt. In diesem Fall wird Lisas Weg jedoch nicht durch Hindernisse mit zeitlich konstanter Position behindert, sondern durch Eisberge, die ihre Position und ihre Ausrichtung ständig verändern. Sie könnten durch Meeresströmungen beschleunigt werden, die mathematisch als Vektorfeld modelliert werden. Auf dieser Grundlage wird die Bewegung der Eisberge als Lösung der Differentialgleichungen in diesem Vektorfeld beschrieben. Dabei wird angenommen, dass Lisa diese Meeresströmungen sehr genau kennt und dem Programm übergeben kann. Als weitere Eingabe sind die Positionen und Anfangsgeschwindigkeiten der Eisberge erforderlich, die im Sachzusammenhang zum Beispiel aus aktuellen Satellitendaten stammen könnten. Das Programm soll Lisa dann den Weg durch das Eis berechnen, bei dem sie zum spätest möglichen Zeitpunkt abreisen kann. Bevor diese Berechnungen vorgenommen werden, muss zunächst eine Physiksimulation die künftigen Positionen der Eisberge berechnen und dabei unter anderem Kollisionen vorhersagen.

Zur eigentlichen Wegberechnung muss man dann die Situation in diskrete Zeitabschnitte unterteilen, von denen man Sichtbarkeiten berechnen kann. Dabei sollte man erwägen, durch geeignete Abschätzungen nur Sichtbarkeiten solcher Wege zu berechnen, die wirklich in Frage kommen, die Berechnung der Wege also mit der Sichtbarkeitsprüfung zu verbinden. Das könnte sich als notwendig erweisen, um zu vermeiden, dass der Rechenaufwand ansonsten bereits bei kleinen Datenmengen so groß wird, dass Lisas Computer den Weg nicht mehr rechtzeitig antreten kann.

4 Beispiele

4.1 Einführung

4.2 Beispiel ohne Hindernis

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Lisa kann die Strasse von ihrem Haus (633.189) direkt erreichen. Dazu
    muss sie 07:28:11 Uhr loslaufen, und die Strasse auf der y-Hoehe
    554 erreichen.
```

4.3 Beispiel 1

Das ist die Darstellung des ersten Beispiels mit Programmausgabe:

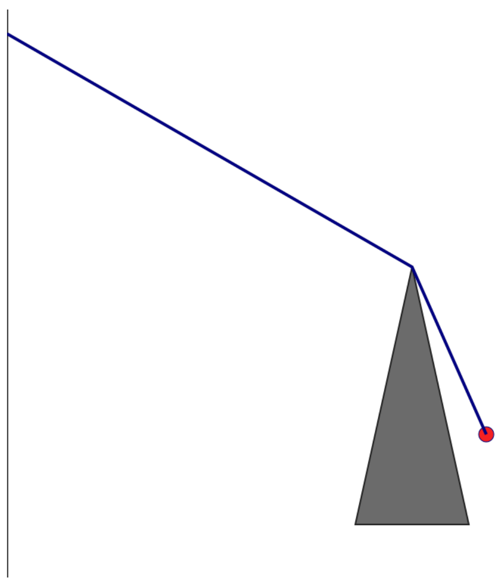

Abbildung 6: Das erste Beispiel

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 2 Wege sollte Lisa von ihrem Haus {633.189} abgehen:
{535.410},
{ 0.718},
Dann muss Lisa spaetestens um 07:27:59 Uhr loslaufen, um 859 Meter
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:03:26 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 3 Knoten gerechnet.
```

```
Dafuer wurde das Schnittgleichungssystem 23-Male geloest.
Insgesamt gibt es 3 Polygomecken.
```

4.4 Beispiel 2

Hier folgt das zweite Beispiel:

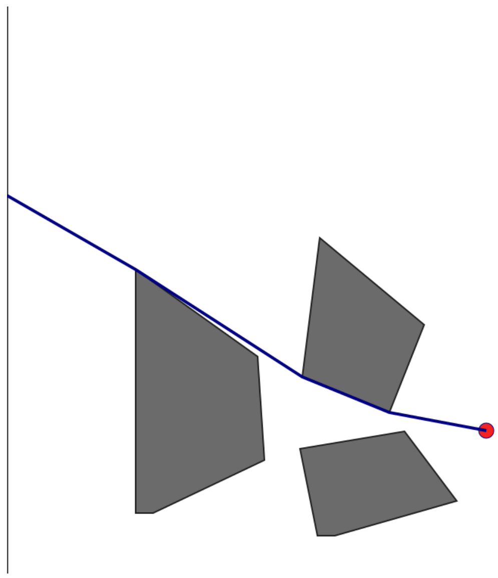

Abbildung 7: Das zweite Beispiel

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 4 Wege sollte Lisa von ihrem Haus {633.189} abgehen:
{505.213},
{390.260},
{170.402},
{ 0.500},
Dann muss Lisa spaetestens um 07:28:08 Uhr loslaufen, um 713 Meter
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:02:51 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 5 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 654-Male geloest.
Insgesamt gibt es 14 Polygomecken.
```

4.5 Beispiel 3

Die Ausgabe für das dritte Beispiel:

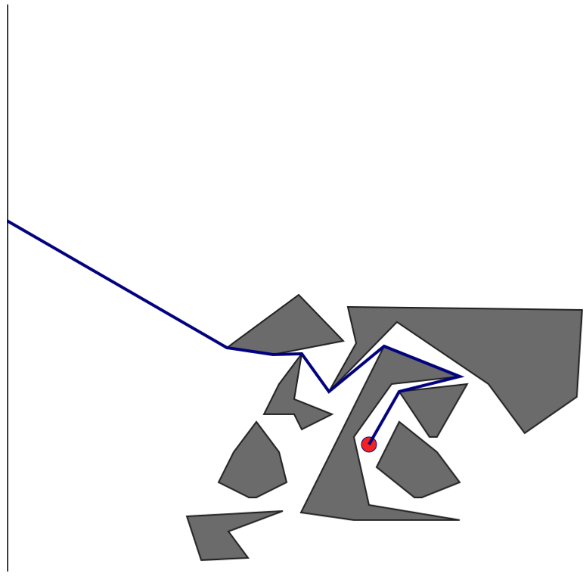

Abbildung 8: Das dritte Beispiel

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 8 Wege sollte Lisa von ihrem Haus {479.168} abgehen:
{519.238},
{599.258},
{499.298},
{426.238},
{390.288},
{352.287},
{291.296},
{ 0.464},
Dann muss Lisa spaetestens um 07:27:28 Uhr loslaufen, um 863 Meter
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:03:27 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 70 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 14137-Male geloest.
Insgesamt gibt es 50 Polygomecken.
```

4.6 Beispiel 4

Die Ausgabe für das vierte Beispiel:

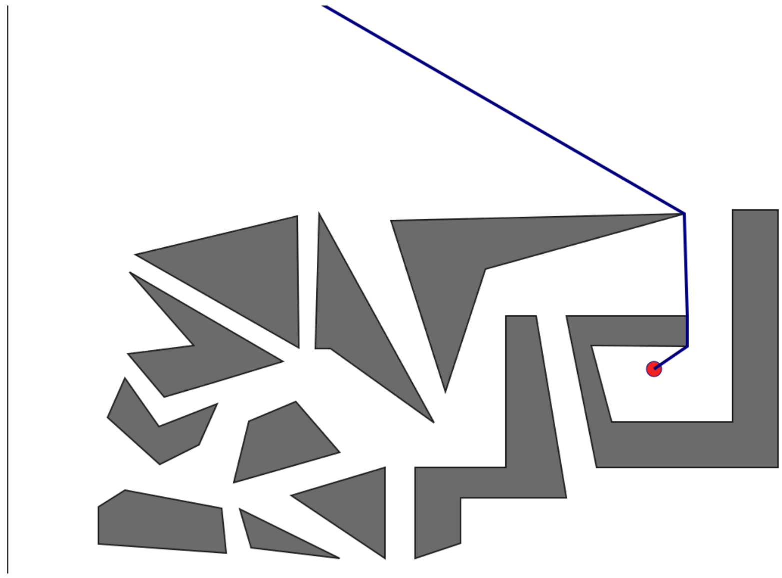

Abbildung 9: Das vierte Beispiel

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 4 Wege sollte Lisa von ihrem Haus {856.270} abgehen:
{900.300},
{900.340},
{896.475},
{ 0.992},
Dann muss Lisa spaetestens um 07:26:55 Uhr loslaufen, um 1263 Meter
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:05:03 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 17 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 25053-Male geloest.
Insgesamt gibt es 56 Polygomecken.
```

4.7 Beispiel 5

Die Ausgabe des Programms für das fünfte Beispiel:

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
```

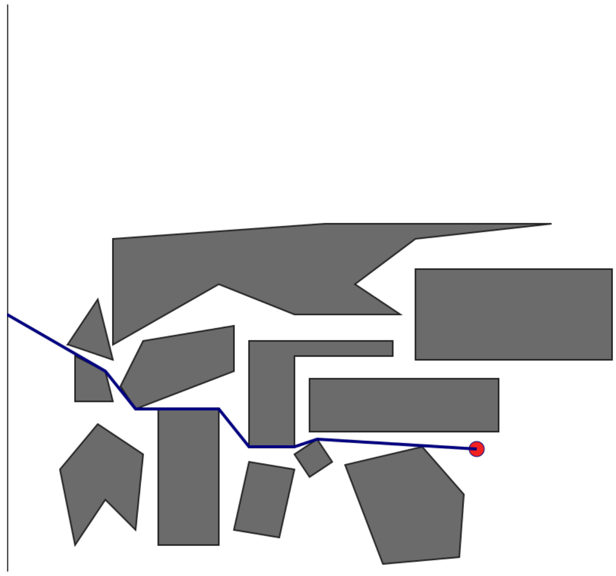

Abbildung 10: Das fünfte Beispiel

```
Diese 8 Wege sollte Lisa von ihrem Haus {621.162} abgehen:
{410.175},
{380.165},
{320.165},
{280.215},
{200.215},
{170.215},
{130.265},
{ 0.340},
Dann muss Lisa spaetestens um 07:27:54 Uhr loslaufen, um 691 Meter
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:02:45 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 142 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 25610-Male geloest.
Insgesamt gibt es 62 Polygomecken.
```

4.8 Beispiel mit 282 Ecken

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 9 Wege sollte Lisa von ihrem Haus {490.300} abgehen:
{485.301},
{455.321},
{450.331},
```

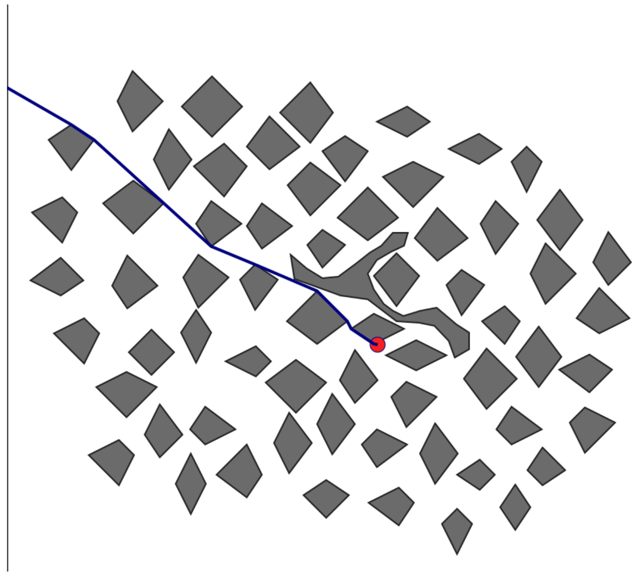

Abbildung 11: eigenes Beispiel mit 282 Ecken

```
{410.371},
{328.406},
{270.430},
{115.571},
{85.591},
{ 0.640},
Dann muss Lisa spaetestens um 07:28:51 Uhr loslaufen, um 605 Meter zurueckzulegen. Fuer den Weg benoetigt sie: 00:02:25 (Stunden:Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg" geschrieben.
Dafuer wurden 148 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 2090850-Male geloest.
Insgesamt gibt es 282 Polygomecken.
```

4.9 Beispiel mit 339 Ecken

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 8 Wege sollte Lisa von ihrem Haus {470.340} abgehen:
{452.354},
{402.380},
{358.414},
{320.428},
{280.448},
{243.474},
{157.532},
```

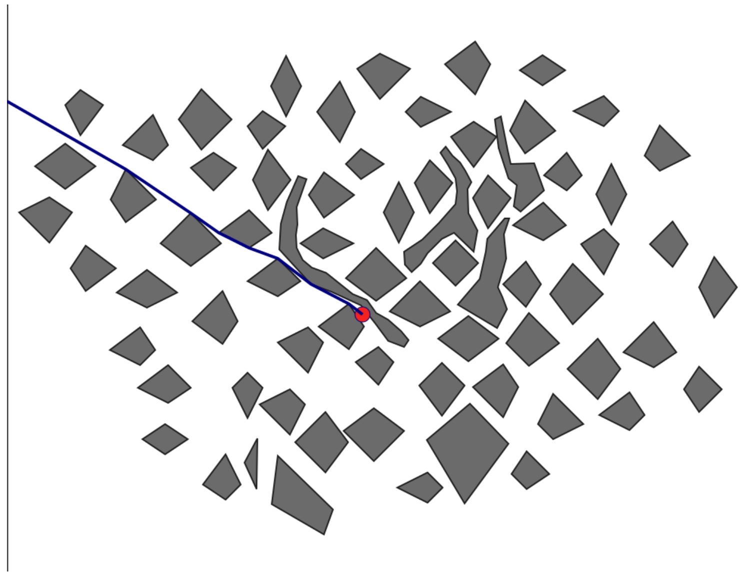

Abbildung 12: eigenes Beispiel mit 339 Ecken

```
{ 0.622},
Dann muss Lisa spaetestens um 07:29:02 Uhr loslaufen , um 550 Meter
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:02:11 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 143 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 3070289−Male geloest.
Insgesamt gibt es 339 Polygomecken.
```

4.10 Beispiel mit 388 Ecken

```
Die Geschwindigkeiten im m/s: 4.167 (Lisa), 8.333 (Bus)
Diese 11 Wege sollte Lisa von ihrem Haus {664.200} abgehen:
{654.210},
{624.230},
{570.240},
{527.295},
{487.325},
{461.323},
{385.363},
{380.373},
{336.410},
{290.419},
{ 0.586},
Dann muss Lisa spaetestens um 07:28:01 Uhr loslaufen , um 787 Meter
```

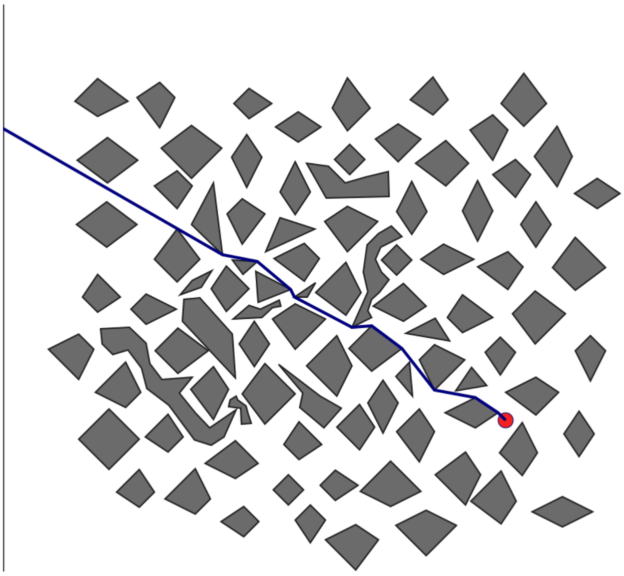

Abbildung 13: eigenes Beispiel mit 388 Ecken

```
    zurueckzulegen. Fuer den Weg benoetigt sie: 00:03:08 (Stunden:
    Minuten:Sekunden)
Eine graphische Darstellung wurde in die Datei "lisa_rennt_result.svg"
    geschrieben.
Dafuer wurden 566 Knoten gerechnet.
Dafuer wurde das Schnittgleichungssystem 4609605-Male geloest.
Insgesamt gibt es 388 Polygomecken.
```

5 Quellcode

Wichtige Ausschnitte aus dem Quellcode sind hier abgedruckt. Dabei ist darauf zu achten, dass wegen Kürzungen die Zeilennummerierung nicht mit der aus dem Originalprogramm übereinstimmt:

```
unsigned long long gaussgeloest = 0;

// Funktion, die prueft, ob und wo zwei Geraden sich schneiden und damit -- wie in der Dokumentation erlaeutert -- eine direkte Verbindung unmoeglich machen
bool treffpunkt(Gerade g1, Gerade g2)
{
    gaussgeloest++;
    // Wenn die Richtungsvektoren Vielfache voneinander sind, ist die Verbindung legitim
    if (vielfache(g1.richtung, g2.richtung) == TRUE)
        return FALSE;

    Bruch var1_bruch;
    Bruch var2_bruch;
    // Abfragen zur Behandlung von speziellen Faellen, bei denen eine Vektorkomponente 0 ist
    if (g1.richtung.deltax == 0)
    {
        var2_bruch.zaehler = g1.ausgangspunkt.x - g2.ausgangspunkt.x;
        var2_bruch.nenner = g2.richtung.deltax;
        var2_bruch = kuerzen(var2_bruch);
        var1_bruch.zaehler = var2_bruch.zaehler * g2.richtung.deltay;
        var1_bruch.nenner = var2_bruch.nenner;
        var1_bruch = addition(var1_bruch, int_to_Bruch(g2.ausgangspunkt.y));
        var1_bruch = addition(var1_bruch, int_to_Bruch(-g1.ausgangspunkt.y));
        var1_bruch = multiplikation(var1_bruch, umkehrung(int_to_Bruch(g1.richtung.deltay)));
    }
    else if (g1.richtung.deltay == 0)
    {
        var2_bruch.zaehler = g1.ausgangspunkt.y - g2.ausgangspunkt.y;
        var2_bruch.nenner = g2.richtung.deltay;
        var2_bruch = kuerzen(var2_bruch);
        var1_bruch.zaehler = var2_bruch.zaehler * g2.richtung.deltax;
        var1_bruch.nenner = var2_bruch.nenner;
```

```
        var1_bruch = addition(var1_bruch, int_to_Bruch(g2.ausgangspunkt.x));
        var1_bruch = addition(var1_bruch, int_to_Bruch(-g1.ausgangspunkt.x));
        var1_bruch = multiplikation(var1_bruch, umkehrung(int_to_Bruch(g1.richtung.deltax)));
    }
    else if (g2.richtung.deltax == 0)
    {
        var1_bruch.zaehler = g2.ausgangspunkt.x - g1.ausgangspunkt.x;
        var1_bruch.nenner = g1.richtung.deltax;
        var1_bruch = kuerzen(var1_bruch);
        var2_bruch.zaehler = var1_bruch.zaehler * g1.richtung.deltay;
        var2_bruch.nenner = var1_bruch.nenner;
        var2_bruch = addition(var2_bruch, int_to_Bruch(g1.ausgangspunkt.y));
        var2_bruch = addition(var2_bruch, int_to_Bruch(-g2.ausgangspunkt.y));
        var2_bruch = multiplikation(var2_bruch, umkehrung(int_to_Bruch(g2.richtung.deltay)));
    }
    else if (g2.richtung.deltay == 0)
    {
        var1_bruch.zaehler = g2.ausgangspunkt.y - g1.ausgangspunkt.y;
        var1_bruch.nenner = g1.richtung.deltay;
        var1_bruch = kuerzen(var1_bruch);
        var2_bruch.zaehler = var1_bruch.zaehler * g1.richtung.deltax;
        var2_bruch.nenner = var1_bruch.nenner;
        var2_bruch = addition(var2_bruch, int_to_Bruch(g1.ausgangspunkt.x));
        var2_bruch = addition(var2_bruch, int_to_Bruch(-g2.ausgangspunkt.x));
        var2_bruch = multiplikation(var2_bruch, umkehrung(int_to_Bruch(g2.richtung.deltax)));
    }
    else
    {
        // Eine 3x2-Matrix wird angelegt, mit der die Loesung der Geradenschnittgleichung mithilfe eines reduzierten Gauss-Algorithmus bestimmt wird
        Bruch gaussmatrix[3][2];
        gaussmatrix[0][0] = int_to_Bruch(g1.richtung.deltax);
        gaussmatrix[0][1] = int_to_Bruch(g1.richtung.deltay);
        gaussmatrix[1][0] = int_to_Bruch(-g2.richtung.deltax);
        gaussmatrix[1][1] = int_to_Bruch(-g2.richtung.deltay);
```

```
        gaussmatrix[2][0] = int_to_Bruch(g2.ausgangspunkt.x - g1.ausgangspunkt.x);
        gaussmatrix[2][1] = int_to_Bruch(g2.ausgangspunkt.y - g1.ausgangspunkt.y);
        // Normierung der ersten Zeile darauf, dass in der ersten Zelle "1" steht
        Bruch zeilenteiler = umkehrung(gaussmatrix[0][0]);
        for (int i = 0; i < 3; i++)
            gaussmatrix[i][0] = multiplikation(gaussmatrix[i][0], zeilenteiler);
        // Elimination der ersten Variable aus der zweiten Zeile
        Bruch eliminationsfaktor = gegenbruch(gaussmatrix[0][1]);
        for (int i = 0; i < 3; i++)
            gaussmatrix[i][1] = addition(gaussmatrix[i][1], multiplikation(gaussmatrix[i][0], eliminationsfaktor));
        // Normierung der zweiten Zeile darauf, dass in der Mitte "1" steht
        zeilenteiler = umkehrung(gaussmatrix[1][1]);
        for (int i = 1; i < 3; i++)
            gaussmatrix[i][1] = multiplikation(gaussmatrix[i][1], zeilenteiler);
        // Elimination der zweiten Variable aus der ersten Zeile
        eliminationsfaktor = gegenbruch(gaussmatrix[1][0]);
        for (int i = 0; i < 3; i++)
            gaussmatrix[i][0] = addition(gaussmatrix[i][0], multiplikation(gaussmatrix[i][1], eliminationsfaktor));
        // Analyse, ob die Geraden sich so schneiden, dass ein Treffpunkt innerhalb des Bereiches vorliegt, in dem sie als Strecken tatsaechliche Bedeutung haben (Siehe Dokumentation fuer Regeln)
        var1_bruch = gaussmatrix[2][0];
        var2_bruch = gaussmatrix[2][1];
    }
    double var1 = bruch_to_double(var1_bruch);
    double var2 = bruch_to_double(var2_bruch);
    if (var1 > 0 && var1 < 1 && var2 > 0 && var2 < 1)
        return TRUE;
    else if ((var2 > 0 && var2 < 1) && (var1 == 0 || var1 == 1))
        return TRUE;
    else if (var2_bruch.zaehler == 0 && var1_bruch.zaehler < var1_bruch.nenner && var1 > 0)
        return TRUE;
    else if (var2_bruch.zaehler == var2_bruch.nenner && var1_bruch.zaehler < var1_bruch.nenner && var1 > 0)
        return TRUE;
```

```
    else
        return FALSE;
}

const unsigned long long A = 1; // Hilfsvariable, die zur Bitmanipulation verwendet wird

int n_hindernisse = 0;                     // tatsaechliche Anzahl der Hindernisse
Hindernis hindernisse[MAXHINDERNISSE]; // Array, in dem alle Hindernisse gespeichert werden
Punkt lisahaus;                            // Der Punkt, an dem Lisas Haus liegt

int ges_punkte;                            // tatsaechliche Anzahl an Ecken der Hindernisse
Punkt punkte[MAXHINDERNISSE * MAXECKEN];   // Array, der alle Ecken der Hindernisse speichert
double rechenmatrix[MAXHINDERNISSE * MAXECKEN][MAXHINDERNISSE * MAXECKEN]; // Speicher, in dem die Weglaengen der Wege zwischen den Ecken der Hindernisse gespeichert sind; IMPOSSIBLE bedeutet, dass der Weg nicht moeglich ist
double haus_zu_punkt[MAXHINDERNISSE * MAXECKEN];       // Speicher, in dem die Weglaengen der Wege von Lisas Haus zu den Eckpunkten gespeichert sind; gleiche Bedeutung fuer IMPOSSIBLE
Ankunft direkt_strasse[MAXHINDERNISSE * MAXECKEN];     // Speicher, der alle notwendigen Informationen ueber die Verbindung der Ecken zur Strasse speichert

Bitmap naechste_punkte_moeglich[MAXHINDERNISSE * MAXECKEN]; // Bitmap, die die Moeglichkeiten von einer Ecke zu anderen als "0" und "1" speichert
Bitmap von_haus_moeglich;                                   // Bitmap, die die Moeglichkeiten vom Haus zu den Ecken speichert

double min_strecke_zu_punkt[MAXHINDERNISSE * MAXECKEN]; // Array,der die kuerzeste bisher bekannte Strecke zu einem Punkt speichert

int v_lisa_km_h = 15;          // Lisas Geschwindigkeit in Kilometern pro Stunde
int v_bus_km_h = 30;           // Busgeschwindigkeit in Kilometern pro Stunde
double v_lisa;                 //Lisas Geschwindigkeit in Metern pro Sekunde
```

```
double v_bus;                    //Busgeschwindigkeit in Metern pro Sekunde
int abfahrtszeitpunkt = 27000; // Bus-Abfahrtszeitpunkt in sek (7:30 Uhr)

// Funktion, die eine Struktur vom Typ "Ankunft" initialisiert
Ankunft init_ankunft()
{
    Ankunft rueckgabe;
    rueckgabe.possible = FALSE;
    rueckgabe.y_pos = 0;
    rueckgabe.entfernung = 0;
    return rueckgabe;
}

// Funktion, die fuer ein einzelnes Hindernis Strecken und Winkel bestimmt
void einzel_hindernis_sw(int i)
{
    for (int j = 0; j < hindernisse[i].n_ecken; j++)
        hindernisse[i].strecken[j] = geradebestimmen(hindernisse[i].ecken[j], hindernisse[i].ecken[(j + 1) % hindernisse[i].n_ecken]);
    for (int j = 0; j < hindernisse[i].n_ecken; j++)
    {
        hindernisse[i].ecken[j].winkelabhaengig = TRUE;
        hindernisse[i].ecken[j].drueber_moeglich_winkel = myatan(hindernisse[i].strecken[j].richtung);
        hindernisse[i].ecken[j].drunter_moeglich_winkel = myatan(vektorumkehren(hindernisse[i].strecken[(j == 0) ? (hindernisse[i].n_ecken - 1) : j - 1].richtung));
    }
}

// Funktion, die die Geradenbeschreibungen zwischen den Ecken in die Strukturen der Hindernisse eintraegt
void hindernisstrecken()
{
    //printf("\nDas sind die Hindernisse mit ihren Punkten und deren Angaben:");
    for (int i = 0; i < n_hindernisse; i++)
        einzel_hindernis_sw(i);
}

// Diese Funktion stellt sicher, dass alle Polygone im Uhrzeitersinn gedreht sind
```

```
void gegen_uhrzeiger()
{
    for (int i = 0; i < n_hindernisse; i++)
    {
        int min_ecke_index = 0;
        for (int j = 1; j < hindernisse[i].n_ecken; j++)
            if (hindernisse[i].ecken[j].y < hindernisse[i].ecken[min_ecke_index].y)
                min_ecke_index = j;
        if (hindernisse[i].ecken[min_ecke_index].drueber_moeglich_winkel < hindernisse[i].ecken[min_ecke_index].drunter_moeglich_winkel)
        {
            //printf("Hindernis %d war im Uhrzeigersinn notiert.", i + 1);
            Hindernis ersatzhindernis;
            ersatzhindernis.n_ecken = hindernisse[i].n_ecken;
            for (int j = 0; j < hindernisse[i].n_ecken; j++)
                ersatzhindernis.ecken[j] = hindernisse[i].ecken[hindernisse[i].n_ecken - j - 1];
            hindernisse[i] = ersatzhindernis;
            einzel_hindernis_sw(i);
        }
    }
}

// Funktion, die die Ecken aus den Hindernisstrukturen in den Array der Punkte eintraegt und ihre Anzahl speichert
void punkteeinfuegen()
{
    ges_punkte = 0;
    for (int i = 0; i < n_hindernisse; i++)
    {
        for (int j = 0; j < hindernisse[i].n_ecken; j++)
        {
            punkte[ges_punkte] = hindernisse[i].ecken[j];
            ges_punkte++;
        }
    }
}

// Funktion, die bestimmt, ob eine Verbindung zwischen zwei Punkten durch die Hindernisse gestoert wird
double verbindungmoeglich(Punkt p, Punkt q)
{
```

```c
    Gerade weg = geradebestimmen(p, q);
    double winkel_von_startpunkt = myatan(weg.richtung);
    double winkel_von_endpunkt = myatan(vektorumkehren(weg.richtung));
    if (p.winkelabhaengig == TRUE && winkel_von_startpunkt < p.drueber_moeglich_winkel && winkel_von_startpunkt > p.drunter_moeglich_winkel && p.drunter_moeglich_winkel < p.drueber_moeglich_winkel)
        return IMPOSSIBLE;
    else if (p.winkelabhaengig == TRUE && (winkel_von_startpunkt > p.drunter_moeglich_winkel || winkel_von_startpunkt < p.drueber_moeglich_winkel) && p.drunter_moeglich_winkel > p.drueber_moeglich_winkel)
        return IMPOSSIBLE;
    else if (q.winkelabhaengig == TRUE && winkel_von_endpunkt < q.drueber_moeglich_winkel && winkel_von_endpunkt > q.drunter_moeglich_winkel && q.drunter_moeglich_winkel < q.drueber_moeglich_winkel)
        return IMPOSSIBLE;
    else if (q.winkelabhaengig == TRUE && (winkel_von_endpunkt > q.drunter_moeglich_winkel || winkel_von_endpunkt < q.drueber_moeglich_winkel) && q.drunter_moeglich_winkel > q.drueber_moeglich_winkel)
        return IMPOSSIBLE;
    else
    {
        for (int i = 0; i < n_hindernisse; i++)
            for (int j = 0; j < hindernisse[i].n_ecken; j++)
                if (treffpunkt(weg, hindernisse[i].strecken[j]) == TRUE)
                    return IMPOSSIBLE;
    }
    return betrag(weg.richtung);
}

// Funktion, die bei Uebergabe eines Punktes gemaess der 30 Grad Regel den Punkt bestimmt, bei dem Lisa bei direkter Verbindung die Strasse traefe
Punkt strasseschnitt(Punkt p)
{
    Punkt strasseerreichen;
    strasseerreichen.x = 0;
    strasseerreichen.y = p.y + (int)((double)p.x * tan(asin(v_lisa / v_bus)));
    return strasseerreichen;
}
```

```

// Funktion, die prueft, ob Lisa von ihrem Haus direkt zur Strasse gehen kann
bool lisadirekt()
{
    Punkt strasseerreichen = strasseschnitt(lisahaus);
    double haus_zu_strasse = verbindungmoeglich(lisahaus, strasseerreichen);
    if (haus_zu_strasse != IMPOSSIBLE)
    {
        double zeit = strasseerreichen.y / v_bus - haus_zu_strasse / v_lisa;
        Zeitpunkt startzeitpunkt = gib_zeitpunkt(zeit + abfahrtszeitpunkt);
        printf("Lisa kann die Strasse von ihrem Haus (%d.%d) direkt erreichen. Dazu muss sie %02d:%02d:%02d Uhr loslaufen, und die Strasse auf der y-Hoehe %d erreichen.\n", lisahaus.x, lisahaus.y, startzeitpunkt.stunden, startzeitpunkt.minuten, startzeitpunkt.sekunden, strasseerreichen.y);
        return TRUE;
    }
    else
        return FALSE;
}

// Diese Funktion fuellt saemtliche Informationen ueber die Wege zwischen den moeglichen Punkten in die dafuer vorgesehenen Arrays ein
void matrixinitialisieren()
{
    // Ab hier wird gespeichert, von welchen Punkten eine 30Grad-konforme Verbindung zur Strasse moeglich ist
    for (int i = 0; i < MAXHINDERNISSE * MAXECKEN; i++)
        direkt_strasse[i] = init_ankunft();
    for (int i = 0; i < ges_punkte; i++)
    {
        Punkt strasseerreichen = strasseschnitt(punkte[i]);
        double testentfernung = verbindungmoeglich(punkte[i], strasseerreichen);
        direkt_strasse[i].entfernung = (testentfernung != IMPOSSIBLE) ? testentfernung : sqrt(pow(strasseerreichen.y - punkte[i].y, 2) + pow(punkte[i].x, 2));
        direkt_strasse[i].y_pos = strasseerreichen.y;
        if (testentfernung != IMPOSSIBLE)
            direkt_strasse[i].possible = TRUE;
```

```
    }
    // Ab hier wird gespeichert, zu welchen Punkten Lisa von ihrem Haus direkt gehen kann; Entfernung wird gespeichert
    for (int i = 0; i < MAXHINDERNISSE * MAXECKEN; i++)
        haus_zu_punkt[i] = 0;
    for (int i = 0; i < ges_punkte; i++)
        haus_zu_punkt[i] = verbindungmoeglich(lisahaus, punkte[i]);
    // Ab hier wird gespeichert, zwischen welchen Ecken der Hindernisse eine direkte Verbindung moeglich ist; Entfernung wird gespeichert
    for (int i = 0; i < MAXHINDERNISSE * MAXECKEN; i++)
        for (int j = 0; j < MAXHINDERNISSE * MAXECKEN; j++)
            rechenmatrix[i][j] = 0;
    for (int i = 0; i < ges_punkte; i++)
    {
        for (int j = i + 1; j < ges_punkte; j++)
        {
            double entfernung = verbindungmoeglich(punkte[i], punkte[j]);
            rechenmatrix[i][j] = entfernung;
            rechenmatrix[j][i] = entfernung;
        }
    }
}

// Auf Grundlage der bereits bekannten Informationen in Rechenmatrix und Array werden hier die Bitmaps initialisiert
void bitboard_initialisieren()
{
    von_haus_moeglich = leereMap();
    for (int i = 0; i < ges_punkte; i++)
        naechste_punkte_moeglich[i] = leereMap();
    for (int i = 0; i < ges_punkte; i++)
        if (haus_zu_punkt[i] != IMPOSSIBLE)
            von_haus_moeglich.einzelspeicher[i / 64] = von_haus_moeglich.einzelspeicher[i / 64] | (A << (i % 64));
    for (int i = 0; i < ges_punkte; i++)
        for (int j = 0; j < ges_punkte; j++)
            if (rechenmatrix[i][j] != IMPOSSIBLE && i != j)
                naechste_punkte_moeglich[j].einzelspeicher[i / 64] = naechste_punkte_moeglich[j].einzelspeicher[i / 64] | (A << (i % 64));
}

// Diese Funktion druckt einen fertig ermittelten Weg zur Strasse
void printsammlung(Wegbeschreibung wb)
```

```c
{
    printf("Diese %d Wege sollte Lisa von ihrem Haus {%d.%d} abgehen:\n", wb.num_anweisungen, lisahaus.x, lisahaus.y);
    for (int i = 0; i < wb.num_anweisungen; i++)
        printf("{%2d.%2d},\n", wb.reihenfolge[i].x, wb.reihenfolge[i].y);
    int gesamtzeit_bis_ankunft = wb.spaeteste_zeit + abfahrtszeitpunkt;
    Zeitpunkt startzeitpunkt = gib_zeitpunkt(gesamtzeit_bis_ankunft);
    Zeitpunkt wegdauer = gib_zeitpunkt(wb.gesamtstrecke / v_lisa);
    printf("Dann muss Lisa spaetestens um %02d:%02d:%02d Uhr loslaufen, um %.0lf Meter zurueckzulegen. Fuer den Weg benoetigt sie: %02d:%02d:%02d (Stunden:Minuten:Sekunden)\n", startzeitpunkt.stunden, startzeitpunkt.minuten, startzeitpunkt.sekunden, wb.gesamtstrecke, wegdauer.stunden, wegdauer.minuten, wegdauer.sekunden);
}

// Diese Funktion schreibt die Umgebung und dem Weg in eine SVG-Datei
void dateischreiben(Wegbeschreibung wb)
{
    FILE *fp = fopen("lisa_rennt_result.svg", "w");
    fprintf(fp, "<svg version=\"1.1\" viewBox=\"0 0 1100 750\" xmlns=\"http://www.w3.org/2000/svg\">\n<g transform=\"scale(1 -1)\">\n<g transform=\"translate(0 -750)\">\n<line id=\"y\" x1=\"0\" x2=\"0\" y1=\"0\" y2=\"750\" fill=\"none\" stroke=\"#212121\" stroke-width=\"3\"/>");
    for (int i = 0; i < n_hindernisse; i++)
    {
        fprintf(fp, "<polygon id=\"P%d\" points=\"", i + 1);
        for (int j = 0; j < hindernisse[i].n_ecken; j++)
            fprintf(fp, "%d %d ", hindernisse[i].ecken[j].x, hindernisse[i].ecken[j].y);
        fprintf(fp, "\" fill=\"#6B6B6B\" stroke=\"#212121\" stroke-width=\"2\"/>\n");
    }
    fprintf(fp, "<circle id=\"L\" cx=\"%d\" cy=\"%d\" r=\"10\" fill=\"#F42121\" stroke=\"#000080\" stroke-width=\"1\"/>\n", lisahaus.x, lisahaus.y);
    fprintf(fp, "<polyline id=\"R2\" points=\"");
    fprintf(fp, "%d %d ", lisahaus.x, lisahaus.y);
    for (int i = 0; i < wb.num_anweisungen; i++)
        fprintf(fp, "%d %d ", wb.reihenfolge[i].x, wb.reihenfolge[i].y);
    fprintf(fp, "\" fill=\"none\" stroke=\"#000080\" stroke-width=\"4\"/>\n</g>\n</g>\n</svg>");
```

```
    printf("Eine graphische Darstellung wurde in die Datei \"lisa_rennt_result.svg\" geschrieben.\n");
}

Wegbeschreibung bester_weg;              // Hier wird der beste bekannte Weg zur Strasse gespeichert
unsigned long long knotengerechnet = 0; // Hier wird die Anzahl der durchgerechneten Knoten gespeichert

// Diese Funktion uebernimmt das Ermitteln des besten Weges und speichert ihn in der oben deklarierten Struktur
void durchrechnen(Wegbeschreibung wege, Bitmap ausgeschlossene, double bisherstrecke, int index, int depth)
{
    // Hier wird die erste Optimierung implementiert: Wenn die beste theoretisch erreichbare Zeit bereits die beste bisher bekannte Zeit ueberschreitet, wird abgebrochen
    if (direkt_strasse[index].y_pos / v_bus - (bisherstrecke + direkt_strasse[index].entfernung) / v_lisa < bester_weg.spaeteste_zeit - 0.01)
        return;
    // Hier wird die dritte Optimierung implementiert: Wenn zu einem Punkt bereits eine kuerzere Strecke bekannt ist, wird abgebrochen; ansonsten wird die neue beste Zeit gespeichert
    if (min_strecke_zu_punkt[index] < bisherstrecke)
        return;
    else
        min_strecke_zu_punkt[index] = bisherstrecke;
    knotengerechnet++;                              // Die Zahl der gerechneten Knoten wird um "1" erhoeht
    wege.reihenfolge[depth] = punkte[index]; // Der aktuelle Punkt wird in die Wegbeschreibung eingetragen
    // Hier folgt die zweite Optimierung
    Bitmap auszuprobierende = bit_and(bm_umkehren(ausgeschlossene), naechste_punkte_moeglich[index]); // Hier werden die Punkte bestimmt, die in der naechsten Rechentiefe auszuprobieren sind
    ausgeschlossene = bit_or(ausgeschlossene, naechste_punkte_moeglich[index]);                   // Hier werden die Punkte bestimmt, die fuer die naechsten Rechentiefen ausgeschlossen werden
    // Wenn vom aktuellen Punkt eine direkte Verbindung zur Strasse moeglich ist, wird der neue Weg als der beste bekannte gespeichert
    if (direkt_strasse[index].possible != FALSE)
    {
        bisherstrecke += direkt_strasse[index].entfernung;
        Punkt strasseerreichen;
```

```
        strasseerreichen.x = 0;
        strasseerreichen.y = direkt_strasse[index].y_pos;
        wege.spaeteste_zeit = strasseerreichen.y / v_bus - bisherstrecke / v_lisa;
        if (wege.spaeteste_zeit > bester_weg.spaeteste_zeit)
        {
            wege.num_anweisungen = depth + 2;
            wege.reihenfolge[depth + 1] = strasseerreichen;
            wege.gesamtstrecke = bisherstrecke;
            bester_weg = wege;
        }
    }
    // Ansonsten werden durch einen rekursiven Aufruf die auszuprobierenden naechsten Punkte ausprobiert
    else
        for (int i = 0; i < ges_punkte; i++)
            if ((auszuprobierende.einzelspeicher[i / 64] & (A << (i % 64))) != 0)
                durchrechnen(wege, ausgeschlossene, bisherstrecke + rechenmatrix[index][i], i, depth + 1);
}

int main(int argc, char *argv[])
{
    v_lisa = ((double)v_lisa_km_h) / 3.6;
    v_bus = ((double)v_bus_km_h) / 3.6;
    printf("Die Geschwindigkeiten im m/s: %.3lf (Lisa), %.3lf (Bus)\n", v_lisa, v_bus);
    if (argc == 1)
    {
        printf("Bitte uebergeben Sie den Dateinamen!\n");
        return 0;
    }
    // Zuerst werden die zur Initialisierung notwendigen Funktionen aufgerufen
    einlesen(argv[1]);
    hindernisstrecken();
    gegen_uhrzeiger();
    // Ab hier wird geprueft, ob eine direkte Verbindung von Lisas Haus zur Strasse moeglich ist
    if (lisadirekt() == TRUE)
        return 0;
    // Wenn das nicht der Fall ist, werden hier die fuer die Wegsuche notwendigen Informationen gesammelt
    punkteeinfuegen();
```

```
    matrixinitialisieren();
    bitboard_initialisieren();
    // Der Speicher, der die minimale Strecke zu einem Punkt und den minimalen Gesamtweg speichert, wird so initialisiert, dass er sinnvoll verwendet werden kann
    for (int i = 0; i < ges_punkte; i++)
        min_strecke_zu_punkt[i] = -STANDARDZEIT;
    bester_weg.spaeteste_zeit = STANDARDZEIT;
    Wegbeschreibung beschreibung;
    beschreibung.spaeteste_zeit = STANDARDZEIT;
    // Von Lisas Haus werden die moeglichen Punkte ausprobiert
    for (int i = 0; i < ges_punkte; i++)
        if (haus_zu_punkt[i] != IMPOSSIBLE)
            durchrechnen(beschreibung, von_haus_moeglich, haus_zu_punkt[i], i, 0);
    // Der beste ermittelte Weg wird gedruckt
    printsammlung(bester_weg);
    dateischreiben(bester_weg);
    printf("Dafuer wurden %llu Knoten gerechnet.\n", knotengerechnet);
    printf("Dafuer wurde das Schnittgleichungssystem %llu-Male geloest.\n", gaussgeloest);
    printf("Insgesamt gibt es %d Polygomecken.\n", ges_punkte);
    return 0;
}
```