Aufgabe 1: Lisa rennt

Teilnahme-ID: 48302

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

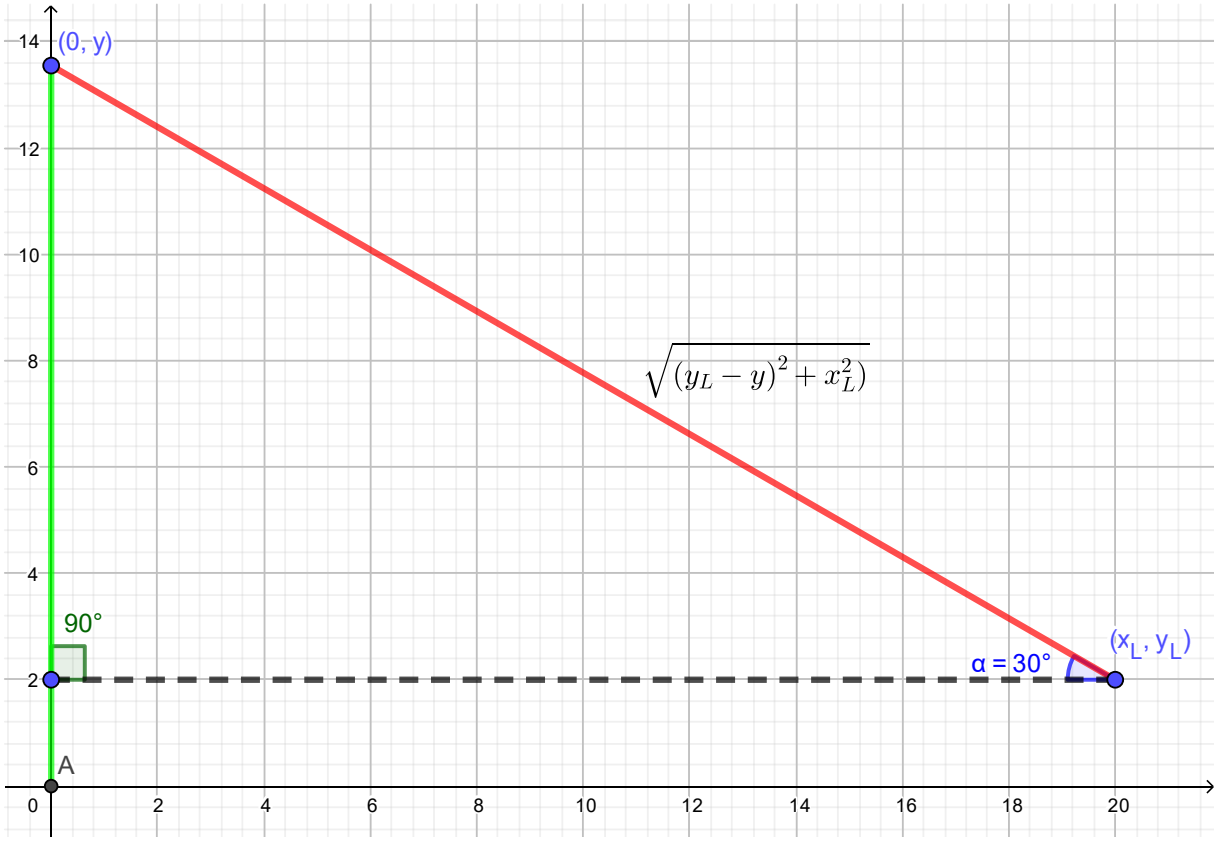
14. April 2019

Inhalt

Lösungsidee

Bester Weg ohne Hindernisse

Um herauszufinden, wo Lisa am besten langgehen sollte, wird zunächst berechnet, wie der beste Weg ohne Hindernisse aussieht. (x_L, y_L) sei die Position von Lisas Haus und $v_B = \frac{25}{3} m/s$ und $v_L = \frac{25}{6}\ m/s$ die Geschwindigkeiten vom Bus und von Lisa in Metern pro Sekunde. Die Strecke, die Lisa zur y-Achse mit einer bestimmten y-Koordinate y laufen müsste, errechnet sich nach Pythagoras mit $\sqrt{(y_L - y)^2 + {x_L}^2}$, also benötigt sie $t = \frac{s}{v} = \frac{\sqrt{(y_L - y)^2 + {x_L}^2}}{v_L}$ Sekunden Zeit. Der Bus benötigt bis zur gegebenen y-Koordinate $\frac{y}{v_B}$ Sekunden. Die folgende Zeichnung aus GeoGebra stellt diese Formeln dar. Lisas Haus befindet sich rechts unten, Lisas Weg ist rot und die Strecke des Busses ist grün.



Die Zeit, zu der Lisa das Haus verlassen muss, berechnet sich aus der Differenz der Zeit, die der Bus braucht, und der Zeit, die Lisa braucht. Diese Zeit ist relativ zum Zeitpunkt, zu dem der Bus von der Haltestelle abfährt, also bedeutet ein negatives Ergebnis, dass Lisa soundso viel Sekunden bevor der Bus abfährt losgehen muss, während ein positives Ergebnis (was nicht häufig vorkommt) bedeutet, dass Lisa sogar nach dem Abfahren des Busses losgehen kann.

Es entsteht die Funktion $f(y) = \frac{y}{v_B} - \frac{\sqrt{(y_L - y)^2 + {x_L}^2}}{v_L}$ mit dem Parameter y, der bekanntermaßen angibt, zu welchem Punkt sich Lisa auf der y-Achse bewegt. Gesucht ist nun das Extremum dieser Funktion, also der späteste Zeitpunkt, der mit einem gewissen Punkt auf der y-Achse erreicht werden kann. Es stellt sich heraus, dass dieser Punkt immer das gleich geformte Dreieck entstehen lässt, bei dem Lisa in einem 30°-Winkel (in der Zeichnung blau eingefärbt) nach oben läuft. Dieser Winkel ist immer vom Verhältnis der beiden Geschwindigkeiten abhängig, denn es gilt $\sin^{-1}\left(\frac{15}{30}\right) =$

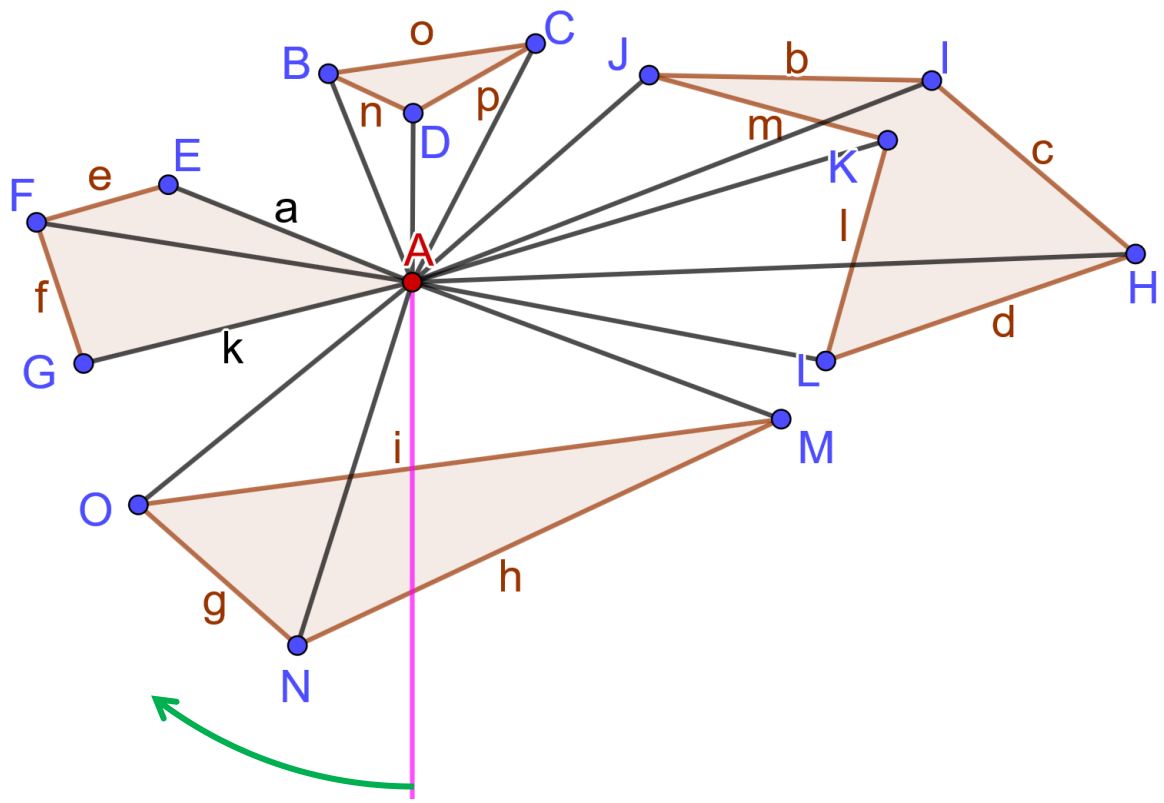
$\sin^{-1}\left(\frac{1}{2}\right) = 30°$. Von einem beliebigen Punkt aus verläuft die bestmögliche Strecke also immer in einem 30° Winkel zur y-Achse, wenn die Geschwindigkeit des Busses doppelt so schnell wie Lisas Geschwindigkeit ist.

Berechnung des Sichtbarkeitsgraphen

Um alle möglichen Wege von Lisas Haus zur y-Achse zu kennen, wird ein sogenannter Sichtbarkeitsgraph (engl. „visibility graph“) erstellt. Ein Sichtbarkeitsgraph verbindet alle Punkte, die sich „sehen“ können, zwischen denen also kein Hindernis in Form einer Kante eines Polygons liegt. Jede Ecke der Polygone wird dafür nicht nur als eine Koordinate, sondern als Knoten („Node“) betrachtet. Einem Knoten werden alle „Nachbar-Knoten“ zugeordnet, also jene Knoten, die von diesem Knoten aus sichtbar sind.

Die Eckpunkte von Lisas Weg bestehen immer aus ihrem Haus, unterschiedlich vielen Eckpunkten von Polygonen und einem Punkt auf der y-Achse, der sich 30° „über“ dem vorletzten Wegpunkt befindet. Andere als diese Punkte kommen im Weg nicht vor, weil Lisa ja an den Polygonen vorbeiläuft, und zwar auf dem besten Weg. Um die Strecke an einem Polygon vorbei möglichst kurz zu halten, muss man die Eckpunkte benutzen. Alles andere wäre nicht der schnellste bzw. beste Weg. Der Sichtbarkeitsgraph umfasst also alle Eckpunkte der Polygone und Lisas Haus und zu jedem Punkt eventuell einen Punkt 30° höher auf der y-Achse, sofern dieser Punkt auf der y-Achse von seinem zugehörigen Punkt aus sichtbar ist.

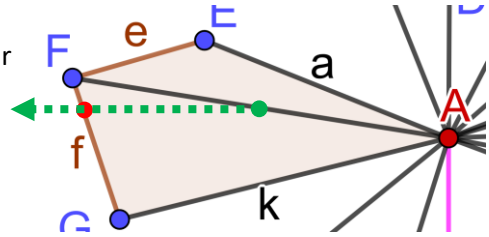
Die wahrscheinlich einfachste Methode, einen solchen Sichtbarkeitsgraphen zu generieren, ist, alle möglichen Punktkombinationen aus zwei Punkten durchzugehen und zu prüfen, ob die Strecke mit irgendeiner Kante kollidiert. Gibt es einen Schnittpunkt mit irgendeiner Kante, dann „sehen“ sich die Punkte nicht. Hier entsteht aber eine Laufzeit von $O(n^3)$, weil jede Punktkombination einmal probiert wird (n^2) und dann mit allen anderen Kanten verglichen wird (n).
Die effizientere Methode heißt „Rotational Plane Sweep“, bei der für alle Punktkombinationen nur in Frage kommende Kanten getestet werden. Es wird jeder Punkt einmal ausgewählt. Von einem ausgewählten Punkt aus werden alle Verbindungen zu den übrigen Punkten, d.h. die Ecken der Polygone, der Punkt von Lisas Haus und der Punkt auf der y-Achse, der 30° über dem gewählten Punkt liegt, betrachtet. Wenn der Punkt auf der y-Achse vom ausgewählten Punkt aus sichtbar ist, wird er als neuer Knoten gespeichert. Folgendes Beispiel mit einer Zeichnung aus GeoGebra veranschaulicht das Vorgehen für einen ausgewählten Punkt allgemein, also ohne Lisas Haus und dem Punkt auf der y-Achse:



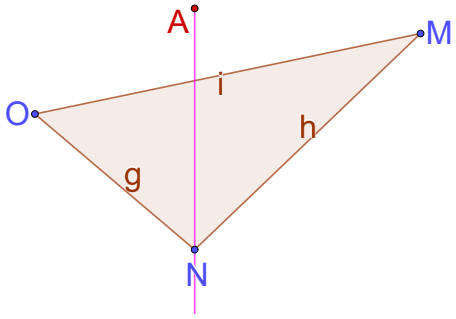
In diesem Beispiel werden alle vom rot dargestellten Punkt A aus sichtbaren Punkte gesucht. Punkt A wird der *Ursprungspunkt* genannt. Es wird für jeden anderen Punkt im Uhrzeigersinn geprüft, ob er vom Ursprungspunkt aus sichtbar ist, wobei bei 0°, also beim pinken Strahl, begonnen wird. Es kommt zuerst Punkt N an die Reihe, danach O und so weiter. Wenn zwei Punkte im selben Winkel zum Ursprungspunkt stehen, wird nach Entfernung sortiert. Es wird eine Liste geführt, die alle Kanten enthält, die für die Sichtbarkeit eines nächsten Punktes relevant sein können. Am Anfang besteht diese Liste aus allen Kanten, die sich mit dem pinken Strahl (Winkel = 0°) schneiden, also hier den Kanten h und i. Beim Punkt N angelangt werden alle Kanten in der Liste durchgegangen und überprüft, ob sich eine von ihnen mit der Strecke $\overline{AN}$ schneidet. Das ist für i der Fall, also ist N nicht sichtbar. An der Kanten-Liste wird nun folgende Veränderung vorgenommen: Die zwei Kanten, bei denen N ein Endpunkt ist (h und g) und die sich bereits in der Liste befinden, werden entfernt. Alle Kanten, die in der Liste fehlen, werden hinzugefügt. Durch diese Veränderung werden alle Kanten, deren zweiter Punkt sich gegen den Uhrzeigersinn befinden (h) und deshalb für die folgenden Punkte nicht mehr relevant sind, entfernt und alle Kanten, deren zweiter Punkt sich im Uhrzeigersinn befindet (g) und deshalb für die folgenden Punkte relevant sein könnte, hinzugefügt. Jedem Knoten müssen daher die zwei Kanten zugeordnet sein, bei denen er ein Endpunkt ist.
Für den Punkt O befinden sich die Kanten i und g in der Kanten-Liste. Keine schneidet $\overline{AO}$, also ist O sichtbar.

Der folgende Punkt, G, gehört zum selben Polygon wie der Ursprungpunkt. Weil die beiden Knoten sich sehen können und sie durch eine Kante (k) verbunden sind, ist keine Überprüfung notwendig und die beiden Knoten sind Nachbarn. Zwischen dem folgenden Punkt F und Ursprungspunkt A befindet sich aber keine Kante, und trotzdem sind sie keine Nachbarn (bei konkaven Polygonen ist es übrigens auch möglich, dass sich zwei Punkte des selben Polygons überhaupt nicht sehen können; in diesem Fall wäre der folgende Test nicht notwendig). Wenn zwei Punkte zum selben Polygon gehören und keine Kante zwischen ihnen liegt, wird geprüft, ob sich die Mitte der verbindenden

Strecke im Polygon befindet. Ist das der Fall, können sich die beiden Punkte doch nicht sehen. Die Mitte der Strecke berechnet sich aus dem Durchschnitt der beiden Werte für die Endpunkte. Um zu prüfen, ob sich ein Punkt in einem Polygon befindet, wird gezählt, wie oft ein Strahl von diesem Punkt aus die Kanten des Polygons schneidet (Abbildung rechts). Wie bei einem Strahl üblich, müsste dieser theoretisch unendlich lang sein, aber hier wird der Strahl so gewählt, dass er nach links parallel zur x-Achse verläuft, und da kein Polygon über die y-Achse ragt, wird der Strahl hier als eine Strecke vom Mittelpunkt zur y-Achse betrachtet. Wenn der Strahl eine gerade Zahl an Kanten des Polygons schneidet, liegt er außerhalb des Polygons, ansonsten innerhalb. Wenn der Punkt auf einer Kante liegt, wird er auch als im Polygon liegend betrachtet.



Zwar kommt es hier nie vor, dass der Mittelpunkt der verbindenden Strecke auf einer Kante liegt, denn für Punkte, die durch eine Kante verbunden sind (z.B. A und G), wird dieser Test überhaupt nicht durchgeführt, aber für die Erweiterung der Aufgabe (s.u.) ist es relevant. Ein spezieller Fall tritt auf, wenn der Strahl zur y-Achse genau durch einen Punkt eines Polygons geht. In diesem Fall würde jede der zwei Kanten, die diesen Punkt berührt, einmal gezählt, obwohl der Strahl das Polygon an dieser Stelle nur einmal schneidet. Im englischsprachigen Wikipedia-Artikel zu diesem Thema steht, es reiche aus, eine Kante nur zu zählen, wenn sich ihr zweiter Punkt unterhalb des Punktes befindet, von dem geprüft werden soll, ob er im Polygon liegt. Das ist jedoch nicht ganz richtig, denn wenn der Punkt auf der oberen Kante eines Rechtecks liegt, wird der Schnittpunkt mit der linken Seite gezählt, liegt er auf der unteren Kante, wird der Schnittpunkt mit der linken Seite nicht gezählt. Die Lösung besteht darin, wenn ein Eckpunkt getroffen wird, getrennt zu zählen, ob der zweite Eckpunkt der Kante oben oder unten liegt. Am Ende werden die beiden Werte verglichen: Sind beide entweder ungerade oder gerade, passiert nichts und es kann ganz normal geprüft werden, ob die Anzahl der Schnittpunkte mit den Kanten (natürlich zuzüglich entweder Schnittpunkte mit Eckpunkten, bei denen der zweite Punkt oberhalb liegt, oder Schnittpunkte mit Eckpunkten, bei denen der zweite Punkt unterhalb liegt) gerade oder ungerade ist. Wenn beide aber verschieden sind, dann liegt der Punkt auf jeden Fall im Polygon (bzw. auf einer Kante).
Trotz dieser anderen Berechnung, um die Sichtbarkeit zu bestimmen, wird die Kanten-Liste dennoch wie gehabt aktualisiert.



Es gibt einen Sonderfall, wenn der pinke Strahl einen Eckpunkt schneidet. Die nebenstehende Zeichnung illustriert diesen Fall, in der N auf dem Strahl liegt.
In diesem Fall dürfen nur g und i, nicht aber h, zur Kanten-Liste hinzugefügt werden.
Würde h doch am Anfang hinzugefügt werden, würde h aus der Liste entfernt werden, sobald Punkt M an der Reihe ist. Das ist aber nicht richtig, denn eigentlich darf h erst hinzugefügt werden, sobald Punkt M an der Reihe ist. Andernfalls würde h für weitere Überprüfungen für Punkte nach M fehlen. Deshalb werden nur Kanten hinzugefügt, deren zweiter Punkt sich im Uhrzeigersinn vom Strahl befindet. Hier wird Punkt N übrigens als letzter Punkt auf Sichtbarkeit überprüft.

Berechnung des besten Wegs – der Dijkstra-Algorithmus

Nachdem im ersten Schritt wie beschrieben der Sichtbarkeitsgraph erstellt wurde, soll im zweiten Schritt der beste Weg gefunden werden. Hierzu eignet sich der Dijkstra-Algorithmus, der allerdings ein bisschen modifiziert werden muss. Eigentlich berechnet der Dijkstra-Algorithmus den kürzesten Weg, hier ist aber nach dem „besten" Weg mit dem spätesten Zeitpunkt für Lisa gefragt. Daher wird jedem Knoten der letztmögliche Zeitpunkt zugeordnet, zu dem Lisa diesen Knoten erreichen muss, um noch rechtzeitig zum Bus zu kommen.

Der Dijkstra-Algorithmus lässt sich folgendermaßen erklären:

1. Es wird eine Liste von allen Knoten, die noch nicht besucht wurden, geführt (also am Anfang alle). Nur den Punkten auf der y-Achse wird ein letztmöglicher Zeitpunkt gemäß der Strecke, die der Bus zu ihnen benötigt, zugewiesen, allen anderen ist als letztmöglicher Zeitpunkt negativ unendlich zugewiesen.
2. Besuche den Knoten k mit der „besten" Zeit aus der Liste der noch unbesuchten Knoten, also in diesem Fall den mit dem spätesten Zeitpunkt, und entferne ihn aus der Liste mit den unbesuchten Knoten. Wenn k der Knoten von Lisas Haus ist, ist die kürzeste Strecke gefunden (siehe unten).
3. Berechne für alle Nachbar-Knoten von k einen neuen letztmöglichen Zeitpunkt von k ausgehend (vom letztmöglichen Zeitpunkt für k wird also die benötigte Zeit für die Strecke zum Nachbar-Knoten abgezogen). Ist die Zeit für einen Nachbar-Knoten besser als vorher, speichere die neue Zeit und speichere k als neuen „Parent"-Knoten für diesen Nachbar-Knoten.
   Gehe zu Schritt 2 zurück.

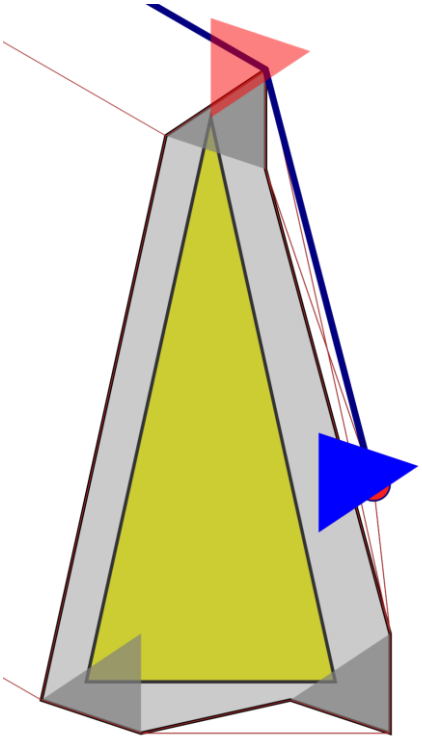
Sollte k in Schritt 2 Lisas Haus sein, ist der beste Weg gefunden und der letztmögliche Zeitpunkt, der im Knoten von Lisas Haus irgendwann in Schritt 3 eingetragen wurde, ist der letztmögliche Zeitpunkt für Lisa, um das Haus zu verlassen. Das funktioniert, weil in Schritt 2 immer der Knoten mit der „besten" Zeit ausgewählt wird. Lisas Haus wird also erst ausgewählt, wenn es wirklich keinen besseren Weg mehr gibt.

Ganz am Anfang wird in Schritt 2 der Knoten, der auf der y-Achse ganz oben liegt, ausgewählt, denn wie in Schritt 1 bereits beschrieben, wird den Knoten auf der y-Achse schon vorher ein letztmöglicher Zeitpunkt entsprechend dem Abstand zum Ursprung $(0|0)$ zugewiesen, der sich durch die Geschwindigkeit des Busses ergibt. Als Zeitpunkt 0 wird dabei 7.30 Uhr festgelegt, also der Zeitpunkt, zu dem der Bus an der Haltestelle abfährt. Ein y-Knoten $(0|10)$ hat demnach einen letztmöglichen Zeitpunkt von $t = \frac{s}{v} = \frac{10m}{8{,}3\overline{\frac{m}{s}}} = 1{,}2s$, wobei $8{,}\overline{3}\,\frac{m}{s}$ die Geschwindigkeit des Busses ist.
Um den vollständigen Weg zu erhalten, wird beginnend mit Lisas Haus immer der Parent-Knoten ausgewählt, solange bis ein Knoten keinen „Parent"-Konten mehr hat, der Knoten also auf der y-Ache liegt.

Erweiterung der Aufgabe

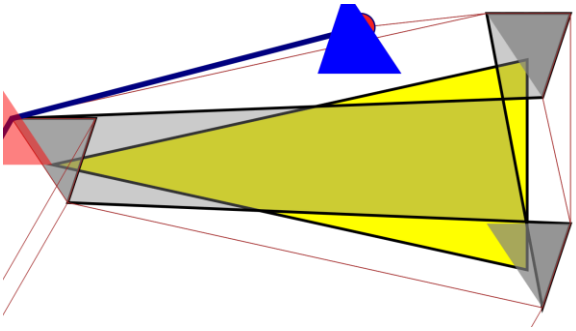
Es ist nicht besonders realistisch, dass ein Mensch durch einen unendlich kleinen Punkt dargestellt wird. Nehmen wir an, statt eines Menschen müsste sich ein Raumschiff in Form eines Dreiecks (vielleicht ja ein Fortbewegungsmittel der Trianguläre aus Aufgabe2?) zur y-Achse bewegen, um einen dort entlanggleitenden Zug mit Waren zu beladen. Das Raumschiff darf dabei nur verschoben,

nicht aber gedreht werden. Um dieses Problem zu lösen, wird nicht etwa statt mit einem unendlich kleinen Punkt mit einem Raumschiff gerechnet, sondern die Polygone werden an den Seiten so vergrößert, dass für die Berechnung immer noch ein Punkt verwendet wird, der berechnete Weg aber für ein sich verschiebendes Polygon gilt. Der entstandene Raum, in dem die Polygone sozusagen „vergrößert“ wurden, nennt sich Konfigurationsraum (engl. „configuration space“). Zur Berechnung des Konfigurationsraumes wird eine einfache, von mir selbst erdachte Methode verwendet, die für viele Beispiele richtige Ergebnisse liefert.



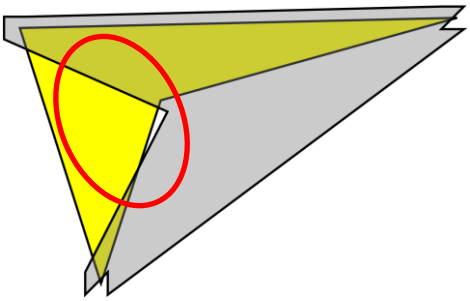
Um das Polygon zu definieren, wird eine zusätzliche Angabe in der letzten Zeile der Beispieldaten-Dateien benötigt. Diese Angabe definiert ein Polygon nach dem festgelegten Schema (zuerst Anzahl der Ecken, dann die Eckpunkte) direkt nach dem Punkt von Lisas Haus. Die nebenstehende Zeichnung gibt ein Beispiel. Rechts ist der rote Punkt von Lisas Haus zu erkennen und das blaue Polygon („Lisas Polygon“) soll durch Verschiebung die y-Achse (hier nicht sichtbar) erreichen. Der Punkt von Lisas Haus dient als Fixpunkt von Lisas Polygon.
Bei dem roten Polygon im Bild handelt es sich um ein Duplikat von Lisas Polygon, um zu zeigen, dass die Grenzen des gelben Dreiecks nicht überschritten werden. Die roten Linien stehen für den Sichtbarkeitsgraphen.

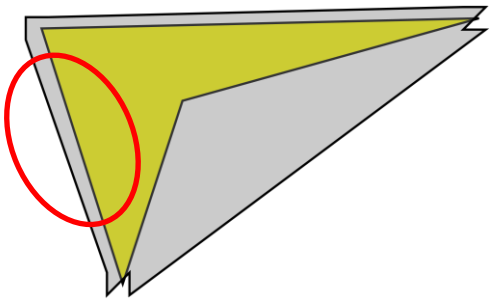
Das Hindernis ist eigentlich das gelbe Polygon, aber durch Vergrößerung ist das graue Polygon entstanden. Die Vergrößerung wird erzeugt, indem Lisas Polygon umgedreht, d.h. um 180° gedreht, an jeder Ecke jedes Hindernis-Polygons mit dem Fixpunkt (Lisas Haus) platziert wird (im Bild grau eingezeichnet). Die einzelnen Ecken der Duplikate von Lisas Polygon werden nun gefiltert. Es bleiben alle Ecken übrig, die sich nicht im originalen, gelben Polygon befinden. Diese Ecken werden dann gemäß dem Winkel vom Mittelpunkt des Polygons aus sortiert, um die Form, die das Polygon vorher hatte, anzunehmen. Ansonsten könnten die einzelnen Ecken unter Umständen nicht in der richtigen Reihenfolge sein, wie folgendes Bild zeigt:



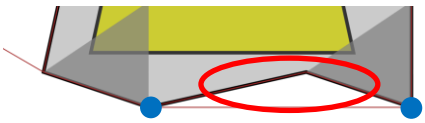
Hier tritt ein Problem auf: Eventuell werden die Ecken nicht korrekt sortiert, wenn das Polygon konkav ist, denn dann könnte zum Beispiel der Mittelpunkt außerhalb des Polygons liegen, wodurch die Eckpunkte falsch sortiert werden würden. Das Problem kann umgangen werden, wenn konkave Polygone vorher in konvexe Polygone zerlegt werden. Außerdem werden alle Punkte des grauen Polygons, die Eckpunkte von zwei im gelben Polygon liegenden Kanten sind, entfernt. Durch diese

Filterung werden Polygone wie im folgenden linken Bild verhindert, die dann stattdessen so wie im rechten Bild aussehen:





Es gibt noch eine andere Schwierigkeit, denn zum Teil wird das entstehende Polygon konkav und nicht wie erwartet konvex:



Die mit einem rot markierten Kreis markierte Ecke müsste eigentlich übersprungen werden. Dass sie nicht übersprungen wird, ist aber erstmal nicht schlimm, denn die beiden Punkte, die eigentlich verbunden werden müssen (blau), sehen sich, und der Weg wird nie einen Umweg in das Polygon nehmen, sondern immer an den äußeren Ecken entlanglaufen. Ein Problem gibt es aber, wenn die Ecke eines anderen Polygons dafür sorgt, dass sich diese beiden Punkte nicht sehen, aber der Weg durch die rot markierte Ecke frei bleibt.

Ein anderes Problem, das auch durch die Vergrößerung von Polygonen entsteht, wird hier fehlerfrei gelöst. Dabei handelt es sich um die Möglichkeit, dass sich zwei Polygone überlappen. In so einem Fall darf der Weg natürlich nicht frei sein, obwohl sich Punkte innerhalb der beiden Polygone vielleicht sehen könnten. Gelöst wird dies, indem jeder Punkt ausgeschlossen wird, der sich innerhalb eines Polygons befindet. Der Knoten von Lisas Haus wird nie ausgeschlossen, denn selbst wenn er in einem Polygon liegt, wird so noch ein Weg aus dem Polygon heraus gefunden. Jedem ausgeschlossenen Punkt werden zwar keine Nachbarn mehr zugeordnet, er muss aber weiterhin durch den Rotational-Plane-Sweep-Algorithmus berücksichtigt werden, um die Kanten-Liste zu aktualisieren.

Umsetzung

Das Programm wurde mit Python 3.6 auf zwei Windows-10-Rechnern und macOS X und unter Linux Mint mit Python 3.5 getestet. Bei richtiger Ausführung traten keine Fehler (auch keine unstimmigen Ergebnisse) auf.

Das Modul `geometry.py` stellt alle nötigen Klassen bereit, um Punkte (`class Point`), Knoten (`class Node`), Strecken (`class LineSegment`), Kanten (`class Edge`) und Polygone (`class Polygon`) darzustellen und Schnittpunkte per `geometry.get_intersection(line1, line2)` zwischen Strecken/Kanten zu berechnen. Der `WaySearcher` speichert alle benötigten Objekte (`Edges`, `Nodes` usw.) in Attributen, dazu siehe den Quellcode. Wichtig ist, dass jeder `Node` die ID des Polygons und das Polygon selbst speichert, zu dem er gehört.

Der `WaySearcher` besitzt hauptsächlich zwei Methoden, nämlich `create_visibility_graph()` und `dijkstra()`.

Einlesen und Darstellung der Daten

Die Daten werden von der statischen Methode `WaySearcher.from_str` eingelesen und dann als Attribute eines `WaySearcher`-Objekts gespeichert. Für Lisas Haus gibt es das Attribut `lisa_node`, das den entsprechenden Knoten speichert. Bei diesem Knoten ist das Attribut `polygon_id` auf `"L"` gesetzt, das die Polygon-ID (hier eigentlich nicht Polygon) angibt. Jeder Knoten speichert außerdem im `polygon`-Attribut das `Polygon`-Objekt, zu dem er gehört, oder `None` bei Lisas Haus und den Knoten auf der y-Achse. Wenn Lisa nicht durch einen Punkt, sondern ein Polygon definiert ist, enthält das Attribut `lisa_polygon` eine Liste aus Punkten. Diese Punkte stellen das Polygon dar, allerdings relativ zu `lisa_node`, weshalb die x- und y-Koordinate von Lisas Haus von den eingelesenen Punkten des Polygons abgezogen werden.
Für jede Zeile, die ein Polygon enthält, ist die statische Methode `Polygon.from_str` zuständig, die aus der Zeile ein `Polygon`-Objekt erstellt.

Der Sichtbarkeitsgraph

Um den Sichtbarkeitsgraphen zu erstellen, wird zunächst die Liste `nodes_outside_polygons` erstellt, die alle Knoten speichert, die nicht ausgeschlossen werden, weil sie in einem Polygon liegen (siehe die Erweiterung der Aufgabe). Dann wird jeder nicht ausgeschlossene `Node` nacheinander als Ursprungspunkt durchgegangen und in Bezug zu allen anderen `Node`s auf Sichtbarkeit überprüft, wie in der Lösungsidee beschrieben. Zu diesen anderen `Node`s kommt immer ein `Node` dazu, der 30° „über“ dem aktuellen `Node` auf der y-Achse liegt. Dieser `Node` wird erzeugt, indem die Methode `get_rotated_on_y_axis` des Ursprungspunkts aufgerufen wird.
Von dem Ursprungspunkt aus zu allen anderen `Node`s wird ein `LineSegment` erstellt; diese werden in der Liste `lines` aufsteigend nach dem Winkel (`LineSegment.angle`) sortiert gespeichert. Der Winkel wird innerhalb von `LineSegment.__init__` mit der `math.atan2`-Funktion berechnet. Diese hat den Vorteil, dass sie im Gegensatz zur normalen Arkustangens-Funktion gleich zwei Parameter mit Vorzeichen entgegennimmt und damit gleich bestimmt, in welchem Quadranten sich der Punkt befindet, um den exakten Winkel zu errechnen. Bei nur einem Parameter gingen die beiden Vorzeichen nämlich verloren. Der Winkel wird dann noch so umgerechnet, dass ein `LineSegment` nach unten auch den Winkel 0° hat, denn mit einer Strecke nach unten (y=0) vom Ursprungspunkt aus wird jetzt das `set` `test_edges` befüllt, welches alle Kanten enthält, die für die Überprüfung der Sichtbarkeit des nächsten Knotens relevant sind. Alle Kanten, die diese Strecke schneiden (nach `geometry.get_intersection`), werden hinzugefügt. Auch der Sonderfall, wenn ein Eckpunkt einer Kante direkt auf dem Strahl liegt, wird berücksichtigt. Sollte ein Endpunkt einer Kante auf dem Strahl liegen, wird diese nur hinzugefügt, wenn die x-Koordinate des anderen Endpunkts kleiner als die x-Koordinate des Schnittpunkts (also des Endpunkts) ist, d.h. die Kante befindet sich auf der Seite im Uhrzeigersinn des Strahls und ist somit für die folgenden Knoten relevant.
Die eben generierte Liste `lines` wird nun durchgegangen. Sofern der Endpunkt der Strecke nicht ausgeschlossen wurde oder er auf der y-Achse liegt (denn Knoten auf der y-Achse werden für jeden Ursprungspunkt ja erst erstellt und sind nicht in der Liste der nicht ausgeschlossenen Knoten enthalten), werden alle relevanten `Edge`s in `test_edges` durchgegangen. Gibt es keinen Schnittpunkt mit einem `Edge`, ist der Knoten sichtbar. Damit den Knoten aber jeweils der Nachbar zugewiesen werden kann, gibt es verschiedene Bedingungen, die nacheinander geprüft werden und von denen mindestens eine erfüllt sein muss:

1. Die erste Bedingung lautet, dass beide Knoten nicht zum selben Polygon gehören dürfen. Dafür wird das `polygon_id`-Attribut der beiden Knoten verglichen.
2. Da die vierte Bedingung mit dem `polygon`-Attribut von einem `Node` arbeitet, wird in dieser zweiten Bedingung vorher geprüft, ob ein `polygon`-Attribut None ist.
3. Nach der dritten Bedingung zählen die beiden Knoten als Nachbarn, wenn sie zwar zum selben Polygon gehören, aber innerhalb des Polygons direkte Nachbarn sind (also gemeinsam an einer Kante beteiligt sind). Dazu wird überprüft, ob sich ein Knoten im Attribut `polygon_neighbors` des anderen befindet. Dieses Attribut ist eine Liste, die die zwei Knoten speichert, die mit diesem Knoten durch eine Kante verbunden sind.
4. Die vierte Bedingung besagt schließlich, dass sich die Mitte der verbindenden Strecke (also der Durchschnitt der Koordinaten der beiden Knoten) nicht im Polygon befinden darf, zu dem die beiden Knoten gehören. Wie die dritte Bedingung ist diese Bedingung also nur relevant, wenn die beiden Knoten zum selben Polygon gehören. Für diese vierte Bedingung ist die `Polygon.point_in_polygon`-Methode zuständig, die wie in der Lösungsidee beschrieben zählt wie oft ein Strahl die Kanten des Polygons trifft.

Ist eine Bedingung erfüllt, wird dem `neighbors`-Attribut (eine Liste, die alle Nachbar-Knoten speichert) des Knotens, der gerade getestet wurde, der Ursprungspunkt angefügt. In die umgekehrte Richtung ist dies nicht nötig, denn auch der Knoten, der gerade getestet wurde, ist einmal als Ursprungspunkt an der Reihe, sofern es sich nicht um einen Knoten auf der y-Achse handelt. Ansonsten würde jedem Knoten jeder Nachbar doppelt zugeordnet werden. Außerdem ist es wichtig, dass dem Knoten, der gerade getestet wird, ein Nachbar zugewiesen wird und nicht umgekehrt, denn die Knoten auf der y-Achse kommen nie als Ursprungspunkte an die Reihe und deshalb muss ihnen jetzt ein Nachbar zugewiesen werden. Das hat zur Folge, dass alle Verbindungen von Knoten auf der y-Achse nur in eine Richtung, nämlich vom Knoten auf der y-Achse aus zur zugehörigen Polygon-Ecke bzw. Lisas Haus, funktionieren. Dabei handelt es sich um die Richtung, die der Dijkstra-Algorithmus später benötigt.
Egal, ob die beiden Knoten Nachbarn sind oder nicht, in jedem Fall wird die `test_edges`-Menge aktualisiert, indem die symmetrische Differenz von `test_edges` und den Kanten, an denen der Endpunkt beteiligt ist, berechnet wird. Die symmetrische Differenz enthält alle Elemente, die nur in einer Menge auftauchen, nicht aber in beiden. Dadurch werden alle Kanten, an denen der Knoten beteiligt ist und die schon in der Menge sind, entfernt, während alle neuen hinzugefügt werden.
Zum Schluss wird noch der `Node` auf der y-Achse hinzugefügt, falls er jetzt Nachbarn – also den Ursprungspunkt – hat. Sollte er keine Nachbarn haben, ist er offensichtlich nicht sichtbar. Für den Dijkstra-Algorithmus wird für diesen `Node` auch der letzte Zeitpunkt bestimmt, zu dem Lisa ihn erreichen kann. Es handelt sich um die Zeit, die der Bus von der Haltestelle benötigt, um zu diesem Knoten zu gelangen, die in `Node.last_time` gespeichert wird.

Der Dijkstra-Algorithmus

Der Dijkstra-Algorithmus in der Methode `WaySearcher.dijkstra` speichert zunächst alle unbesuchten Nodes (also alle) in der Liste `unvisited_nodes`. Der `Node` mit der spätesten Zeit (`Node.last_time`) wird ausgewählt, alle Zeiten zu den Nachbarn werden mit der Methode `Node.reload_last_time` neu berechnet und sollte die Zeit besser sein, wird sie von der Methode übernommen und der aktuell ausgewählte `Node` wird im `parent`-Attribut gespeichert.
Zum Schluss wird der Weg mithilfe des `parent`-Attributs von Lisas Haus aus zurückverfolgt und die Punkte werden mitsamt der Länge des Wegs und der benötigten Zeit zurückgegeben.

Verbesserungsmöglichkeit

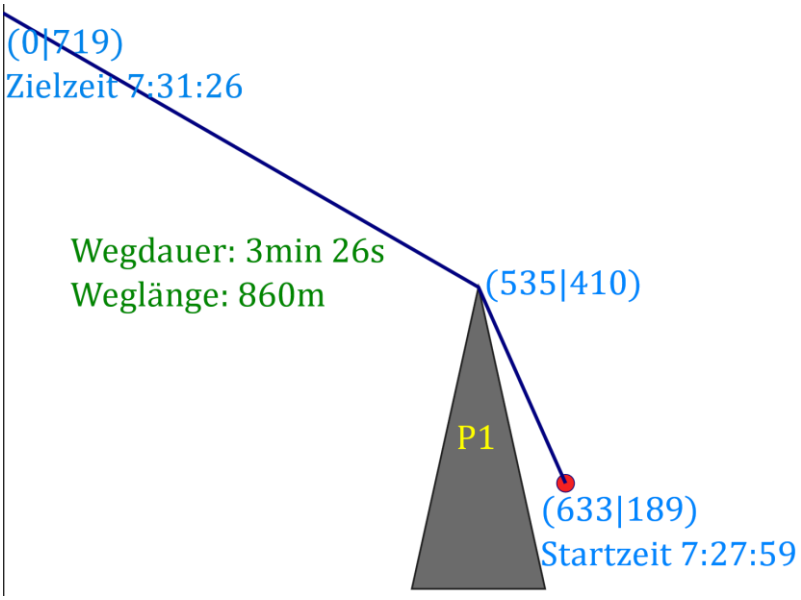
Statt den Sichtbarkeitsgraphen schon am Anfang komplett zu berechnen, besteht die Möglichkeit, ihn stattdessen nur „auf Anfrage“ zu erstellen. Immer, wenn der Dijkstra-Algorithmus Nachbarn zu einem `Node` benötigt, werden diese mit dem Rotational-Plane-Sweep-Algorithmus berechnet. Am Anfang müssten aber alle Nodes auf der y-Achse gefunden werden, um Startpunkte bereitzustellen. Das würde aber oft einen größeren Aufwand bedeuten als den gesamten Sichtbarkeitsgraphen zu erstellen. Höchstens für Gebiete mit sehr vielen Polygonen, die auf Lisas Weg keine Rolle spielen, würde es Sinn ergeben, diese Technik anzuwenden. Dies ist aber in keinem der Beispiele der Fall. Ein Nachteil ist auch, dass am Anfang natürlich beim Knoten ganz oben auf der y-Achse begonnen wird, obwohl noch nicht bekannt ist, ob die zu diesem Knoten zugehörige Polygon-Ecke bzw. Lisas Haus überhaupt sichtbar ist. So kommt es vor, dass am Anfang auf sehr viele Knoten auf der y-Achse der Rotational-Plane-Sweep-Algorithmus angewandt wird, obwohl diese eigentlich überhaupt nicht sichtbar sind und deshalb ignoriert werden.

Darstellung als SVG

In Aufgabe1/aufgabe1.py wird für jede Beispieldaten-Datei der beste Weg berechnet. Dieser beste Weg wird in die SVG-Beispieldaten eingefügt, indem die Zeile gesucht wird, in der `polyline` vorkommt. Diese Zeile wird durch eine `polyline` mit allen Wegpunkten ersetzt.
Außerdem existiert die Methode `WaySearcher.save_svg`, die eine `svg`-Datei mit dem Drittanbieter-Modul `svgwrite` unter der MIT-Lizenz erstellt. Da das in der Aufgabenstellung nicht verlangt wird, wird diese Methode hier nicht weiter besprochen, aber in den Beispielen kommt sie zum Einsatz, um detailreiche Ausgaben (inklusive Sichtbarkeitsgraph) zu erhalten. Die Methode ist noch vor der Veröffentlichung der Beispieldaten mit den entsprechenden SVG-Dateien entstanden. Das `svgwrite`-Modul liegt im Ordner `Aufgabe1/site-packages` bei und sollte vom Programm automatisch gefunden werden.

Beispiele

lisarennt1.txt



Ausgabe:

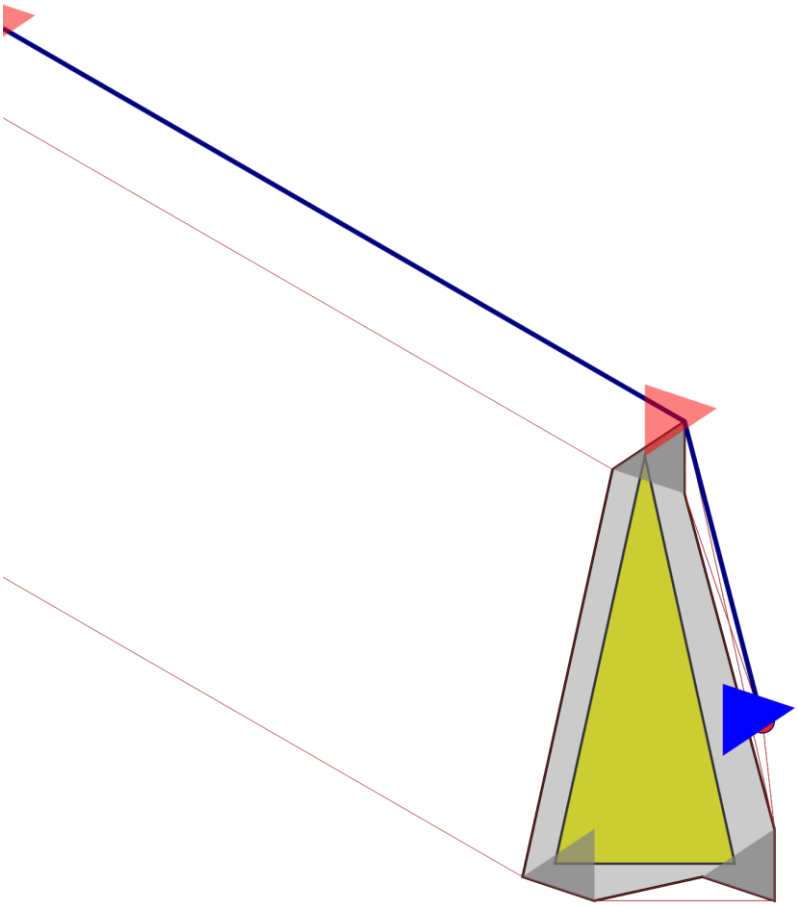
```
Vis. Graph time: 0.0002912046943154859
Dijkstra   time: 4.653800050165226e-05

Startzeit: 7:27:59 Uhr (abgerundet)
Zielzeit:  7:31:26 Uhr (abgerundet)
y-Koord:   719
Wegdauer:  3 Minuten 26 Sekunden (abgerundet)
Weglänge:  860m
WEG:
Starte beim Haus (633|189) (ID: 'L')
gehe zu (535|410) (ID: 'P1')
gehe zu (  0|719) (ID: keine (Endpunkt))
```

Dieses sehr einfache Beispiel ist gut zum Ausprobieren geeignet. Der Dijkstra-Algorithmus benötigt sogar nur etwa 0,00004653 Sekunden (es gibt ja auch nicht viele Möglichkeiten für den Weg).
Hier zeigt sich, dass in vielen Fällen der Weg nach oben die beste Wahl ist, weil ja auch der optimale Weg ohne Hindernisse in einem 30° Winkel nach oben verläuft.
Hier kann auch veranschaulicht werden, dass anderen Winkel als 30° (hier ab dem Punkt $(535|410)$) eine frühere Startzeit bedeuten würden:
Zwischen Lisas Haus $(633|189)$ und dem ersten Wegpunkt $(535|410)$ liegen $\sqrt{(633-535)^2+(189-410)^2} = \sqrt{58445} \approx 241{,}75$ Meter. Das bedeutet für Lisa eine Zeit von etwa $\frac{241{,}75m}{4{,}1\bar{6}m/s} = 58{,}02$ Sekunden. Für die zweite Strecke sind es etwa $\frac{617{,}82m}{4{,}1\bar{6}m/s} = 148{,}28$ Sekunden, was insgesamt tatsächlich 3 Minuten und 26 Sekunden (206 Sekunden) ergibt. Der Bus benötigt hier eine Zeit von $\frac{719m}{8{,}\bar{3}m/s} = 86{,}28$ Sekunden, was bedeutet, dass Lisa 86,28 Sekunden einsparen kann, also muss sie etwa $206s - 86s = 120$ Sekunden, bevor der Bus von der Haltestelle abfährt, losgehen.

Würde Lisa nun zum Punkt $(0|700)$ laufen, würde daraus eine Zeit für die zweite Strecke von $\frac{608{,}54m}{4{,}1\bar{6}m/s} = 148{,}05$ Sekunden folgen. Der Bus würde $\frac{700m}{8{,}\bar{3}m/s} = 84$ Sekunden benötigen und deshalb müsste Lisa $(148{,}05s + 58{,}02s) - 84s = 122{,}07$ Sekunden, bevor der Bus abfährt, losgehen. Hier würde Lisa für ihren Weg zwar weniger Zeit benötigen, dafür fährt der Bus aber früher an der Stelle vorbei, die sie erreichen würde. Das Ganze mit $(0|750)$ sähe dann so aus: Lisa benötigt für die zweite Strecke etwa $\frac{633{,}90m}{4{,}1\bar{6}m/s} = 152{,}14$ Sekunden. Den Bus erreicht sie nun nachdem dieser schon $\frac{750m}{8{,}\bar{3}m/s} = 90$ Sekunden gefahren ist. Es ergibt sich eine Zeit von $(152{,}14s + 58{,}02s) - 90s = 120{,}16$ Sekunden, was auch bei Rechnung mit exakten Werten (sowohl oben als auch hier) mehr ist.

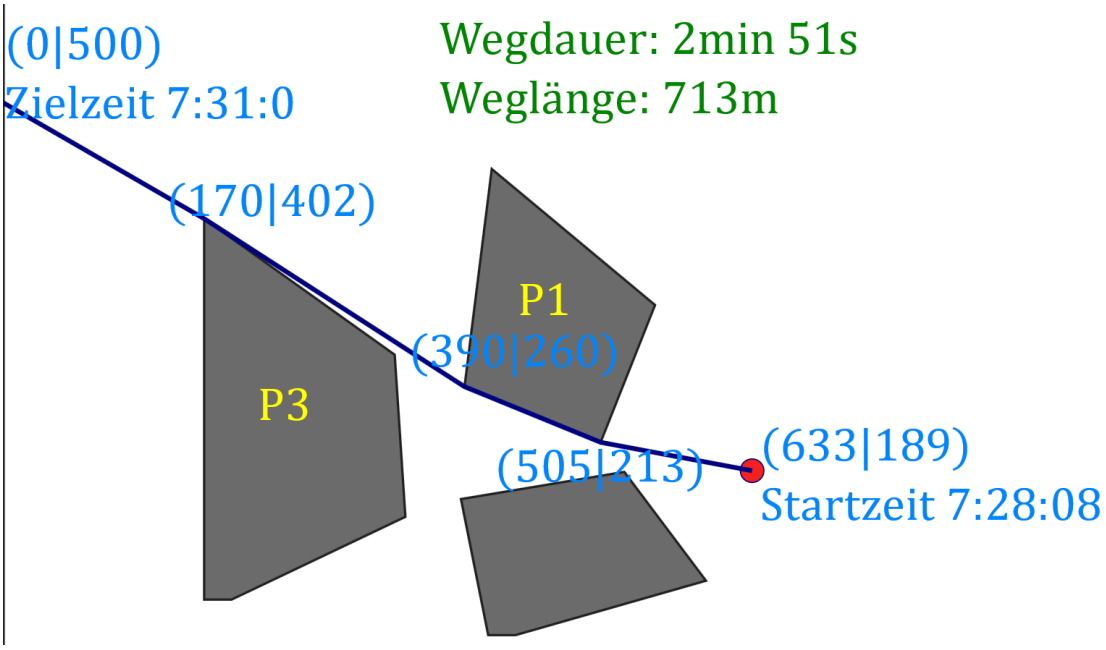
Erweiterung der Aufgabe

Hier wird versucht, statt eines Punktes ein Dreieck zur y-Achse laufen zu lassen. Das Ergebnis sieht so aus:



Hier funktioniert die Erweiterung sehr gut, denn es gibt auch nur ein konvexes Dreieck. Die roten Dreiecke werden jeweils an den Eckpunkten des Weges gezeichnet, um zu zeigen, dass das blaue Dreieck bei Bewegung tatsächlich nicht anstößt. Die Dellen im grauen Polygon wurde bereits in der Lösungsidee besprochen.

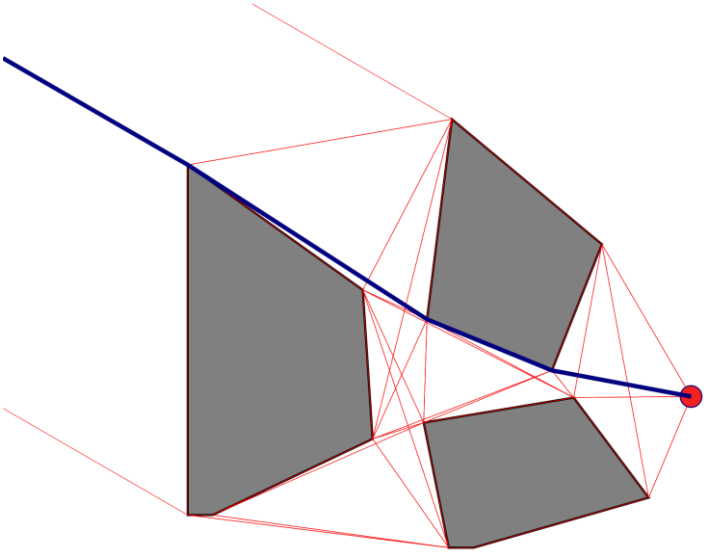
lisarennt2.txt



Ausgabe:

```
Vis. Graph time: 0.004476545018842756
Dijkstra   time: 0.00018546761964628733

Startzeit: 7:28:08 Uhr (abgerundet)
Zielzeit:  7:31:00 Uhr (abgerundet)
y-Koord:   500
Wegdauer:  2 Minuten 51 Sekunden (abgerundet)
Weglänge:  713m
WEG:
Starte beim Haus (633|189) (ID: 'L')
gehe zu (505|213) (ID: 'P1')
gehe zu (390|260) (ID: 'P1')
gehe zu (170|402) (ID: 'P3')
gehe zu (  0|500) (ID: keine (Endpunkt))
```

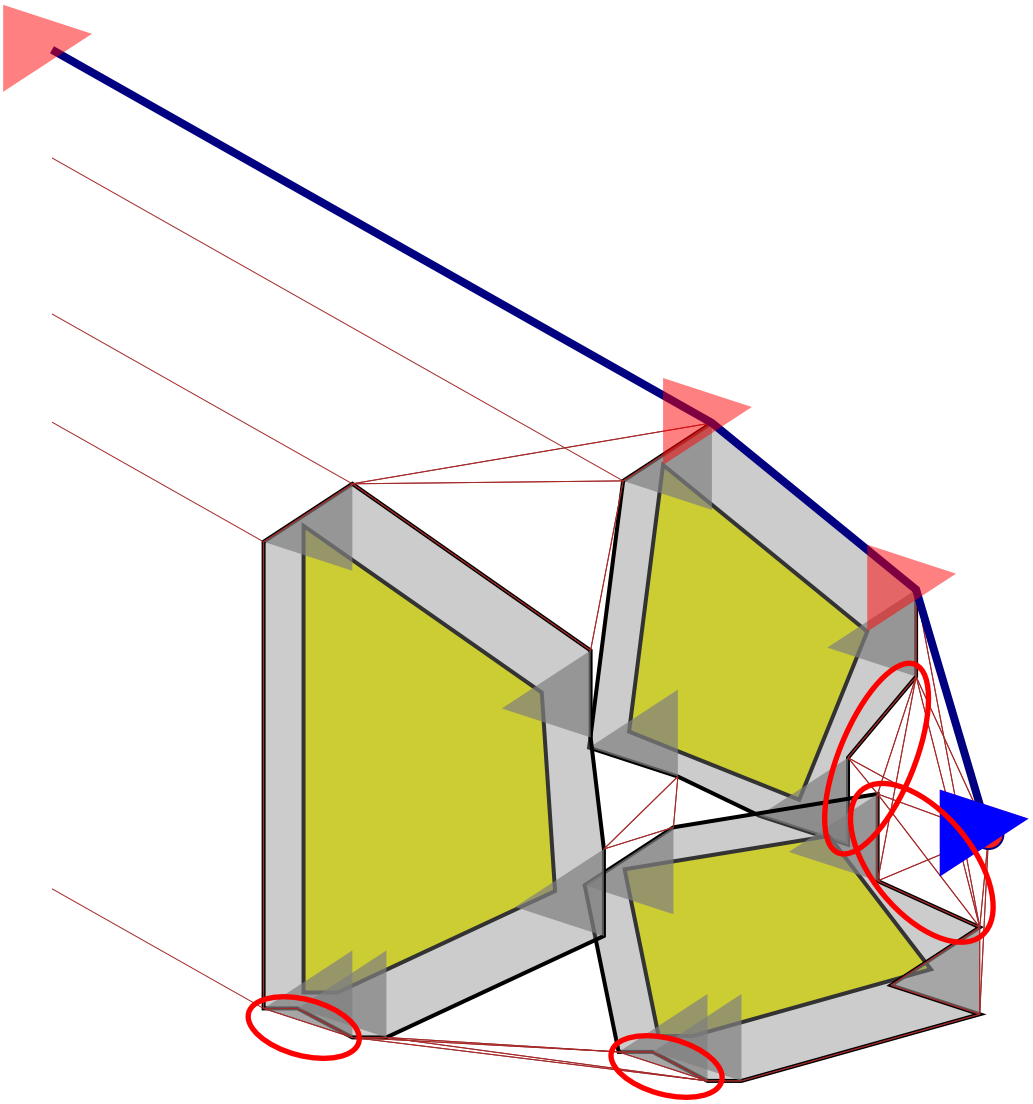
In diesem Beispiel geht der beste Weg unter Polygon `P1` hindurch. Das liegt daran, dass der hiermit entstehende Weg dem 30°-Weg sehr ähnelt und damit der beste Weg ist.



Dieses Beispiel eignet sich außerdem gut, um den erstellten Sichtbarkeitsgraphen zu visualisieren. Nebenstehende SVG-Grafik wurde mit der Methode `WaySearcher.save_svg` erstellt. Der Sichtbarkeitsgraph ist durch die roten Linien dargestellt. Wie hier zu sehen ist, haben alle Knoten auf der y-Achse nur jeweils eine Verbindung, nämlich zu ihrem zugehörigen Punkt 30° „höher“. Alle Polygone haben auf ihren Kanten jeweils auch rote Linien, die die Verbindungen zwischen den Ecken darstellen.

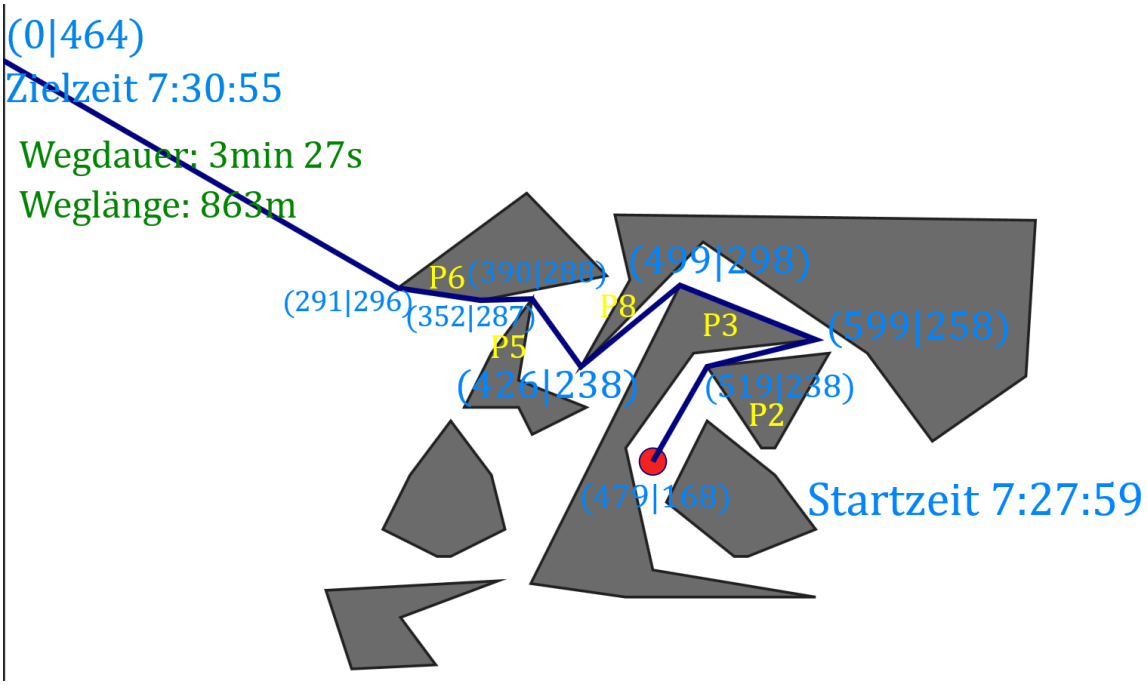
Erweiterung der Aufgabe

Es soll wieder der Weg mit dem gleichen Dreieck wie in Beispieldaten Nummer 1 gefunden werden, das Ergebnis sieht so aus:



Auch hier sind wieder mehrere „Dellen“ (rote Ellipsen) zu erkennen. Wie schon im ersten Beispiel sind diese nicht problematisch. Interessant ist noch, dass der Weg durch zu große Polygone versperrt bleibt. Aus diesem Grund befindet sich nicht auf jeder Kante der Polygone eine rote Linie, weil sich die entsprechenden Ecken durch ein anderes, hereinragendes Polygon nicht sehen können.

lisarennt3.txt



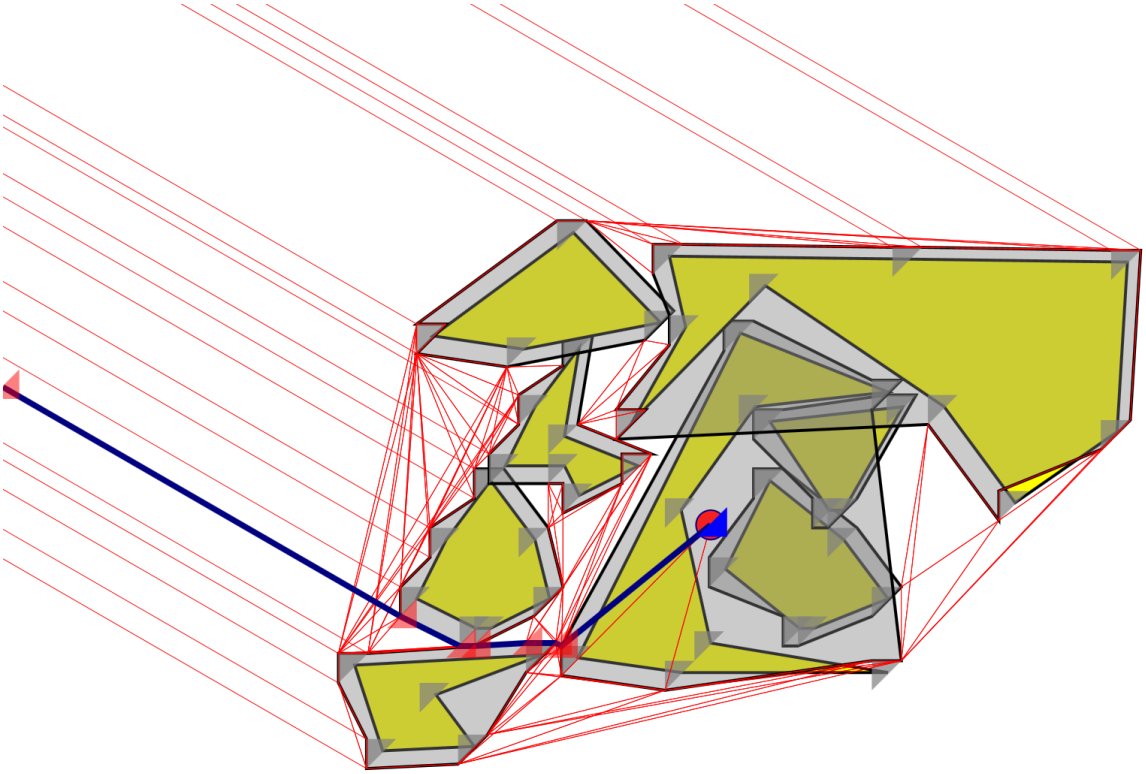
Ausgabe:

```
Vis. Graph time: 0.06750918697770933
Dijkstra   time: 0.0011737157479460858

Startzeit: 7:27:28 Uhr (abgerundet)
Zielzeit:  7:30:55 Uhr (abgerundet)
y-Koord:   464
Wegdauer:  3 Minuten 27 Sekunden (abgerundet)
Weglänge:  863m
WEG:
Starte beim Haus (479|168) (ID: 'L')
gehe zu (519|238) (ID: 'P2')
gehe zu (599|258) (ID: 'P3')
gehe zu (499|298) (ID: 'P3')
gehe zu (426|238) (ID: 'P8')
gehe zu (390|288) (ID: 'P5')
gehe zu (352|287) (ID: 'P6')
gehe zu (291|296) (ID: 'P6')
gehe zu (  0|464) (ID: keine (Endpunkt))
```

Dieses Beispiel ist vergleichsweise eines der komplexeren. Hier muss Lisa sogar ein Stück zurückgehen, um am Polygon `P3` vorbeizukommen. Das ist auch ein Vorteil der Verwendung des Dijkstra-Algorithmus gegenüber irgendwelchen anderen Verfahren, die einen Weg suchen, ohne alle Möglichkeiten per Sichtbarkeitsgraph zu berücksichtigen, denn für den Dijkstra-Algorithmus ist es nicht relevant, ob Lisa ein Stück zurückgeht oder nicht. Auch hier ist der Weg wieder dem 30°-Weg sehr ähnlich.

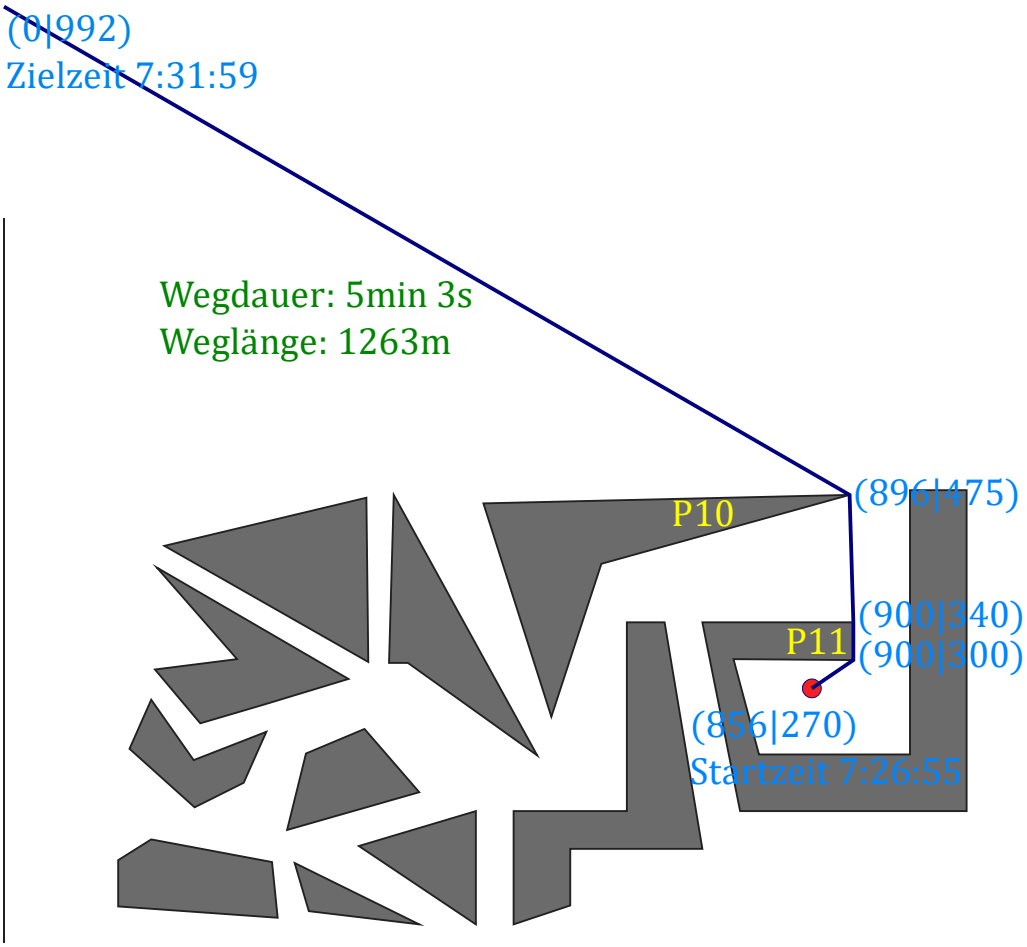
Erweiterung der Aufgabe

Mit einem ähnlichen Dreieck wie in den Beispieldaten Nummer 1 und 2 wird versucht, einen Weg zu finden:



Der Weg ist hier nicht ganz so gut, weil Lisas Haus von einem konkaven Polygon umschlossen ist. Das konkave Polygon wird durch das Filtern der Punkte, die an zwei Kanten anliegen, von denen beide den gelben Teil des Polygons schneiden, weiter verbessert. Korrekterweise wird der Knoten von Lisas Haus nicht ausgeschlossen, sondern stattdessen werden Verbindungen zu den anderen Punkten hergestellt. Dadurch führt der Weg auch aus dem Polygon heraus, aber die Umrisse des Polygons werden trotzdem nicht besonders gut berechnet, weil es sich eben um ein konkaves Polygon handelt. Das wäre nicht so schlimm, wenn Lisas Haus nicht auch noch von diesem Polygon umschlossen wäre. Lösen kann man dieses Problem im allgemeinen, indem vorher konkave Polygone in konvexe Polygone zerlegt werden.

lisarennt4.txt



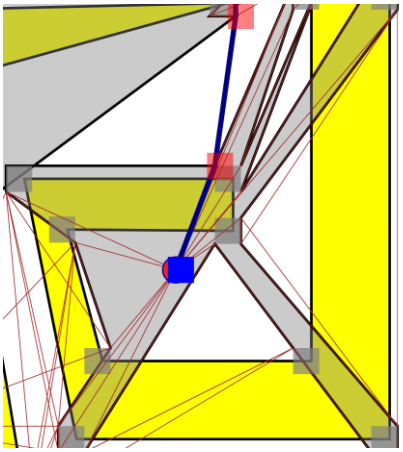
Ausgabe:

```
Vis. Graph time: 0.11391407511027968
Dijkstra   time: 0.001712666856696854

Startzeit: 7:26:55 Uhr (abgerundet)
Zielzeit:  7:31:59 Uhr (abgerundet)
y-Koord:   992
Wegdauer:  5 Minuten 3 Sekunden (abgerundet)
Weglänge:  1263m
WEG:
Starte beim Haus (856|270) (ID: 'L')
gehe zu (900|300) (ID: 'P11')
gehe zu (900|340) (ID: 'P11')
gehe zu (896|475) (ID: 'P10')
gehe zu (  0|992) (ID: keine (Endpunkt))
```

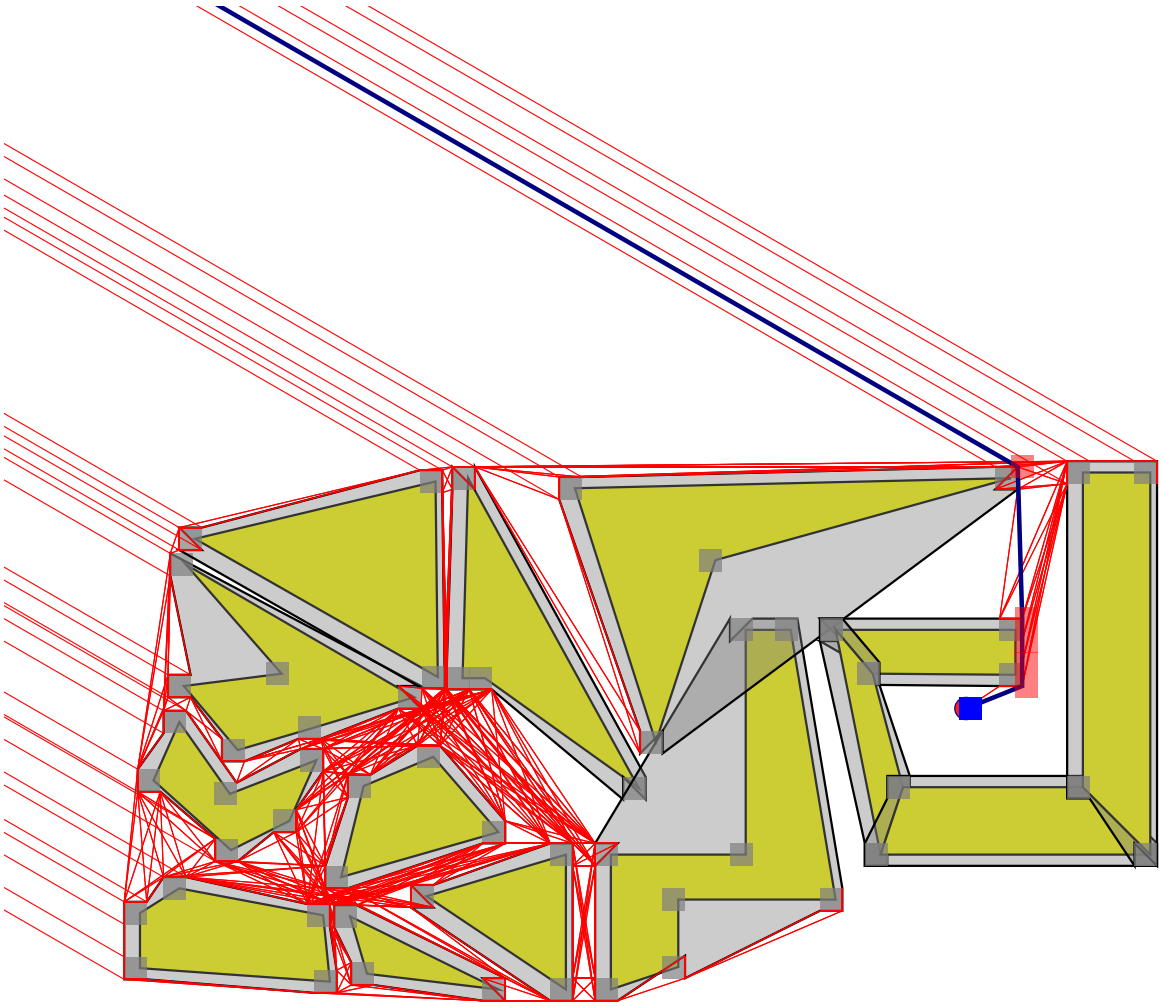
Hier ist ein eventuelles Anwendungsbeispiel der Verbesserungsmöglichkeit von oben zu sehen. Es gibt sehr viele Polygone, von denen man eigentlich den Sichtbarkeitsgraph nicht hätte generieren müssen. Das ist dann wohl auch der Grund für die vergleichsweise „lange“ Zeit von 0,1 Sekunden, aber die Zeit, um alle `Nodes` auf der y-Achse zu finden, wäre vermutlich nicht kürzer.

Erweiterung der Aufgabe

Diesmal wird versucht, mit einem Quadrat einen Weg zu finden. Hier ist nur ein Ausschnitt des Ergebnisses dargestellt:

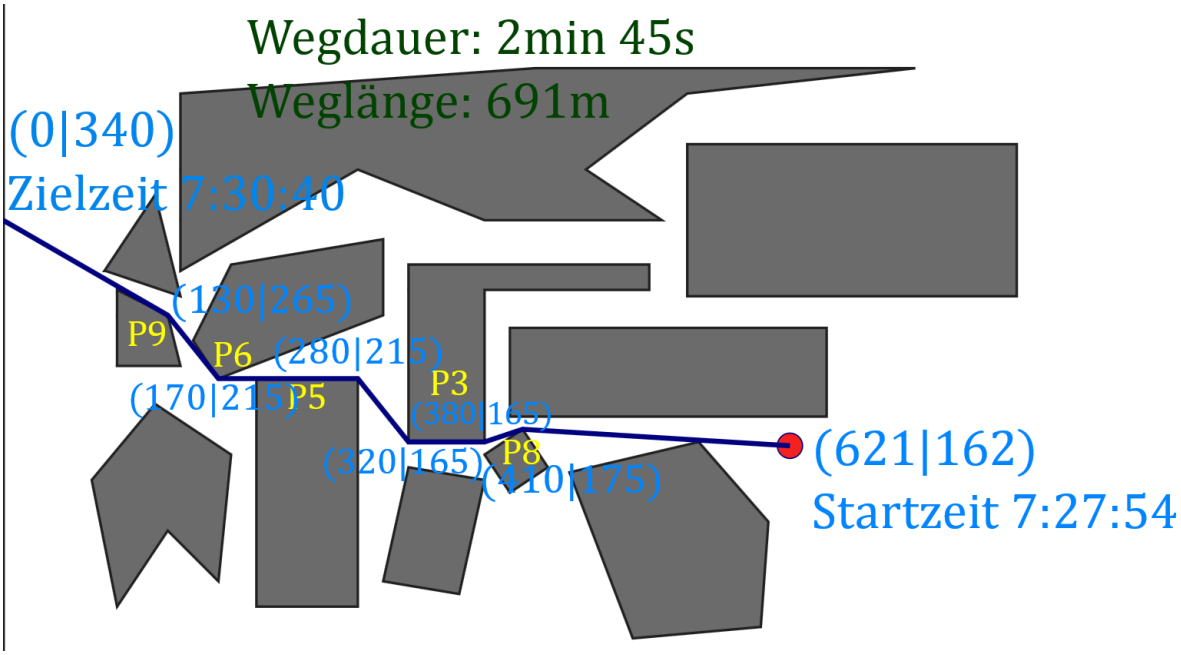


Auch hier entsteht durch das konkave Polygon wieder ein Problem, das sich lösen lässt, indem das Polygon in konvexe Polygone aufgeteilt wird. Für die folgende Grafik wurde deshalb das Polygon P11 vorher in vier kleinere Teile zerlegt:



Der korrekte Weg wird fehlerfrei gefunden. Grundsätzlich könnte der Algorithmus also verbessert werden, wenn alle konkaven Polygone vorher in konvexe zerlegt werden. Das Filtern der Punkte, die Eckpunkte von zwei Kanten sind, die das gelbe Polygon schneiden, verbessert auch hier die konkaven Polygone.

lisarennt5.txt



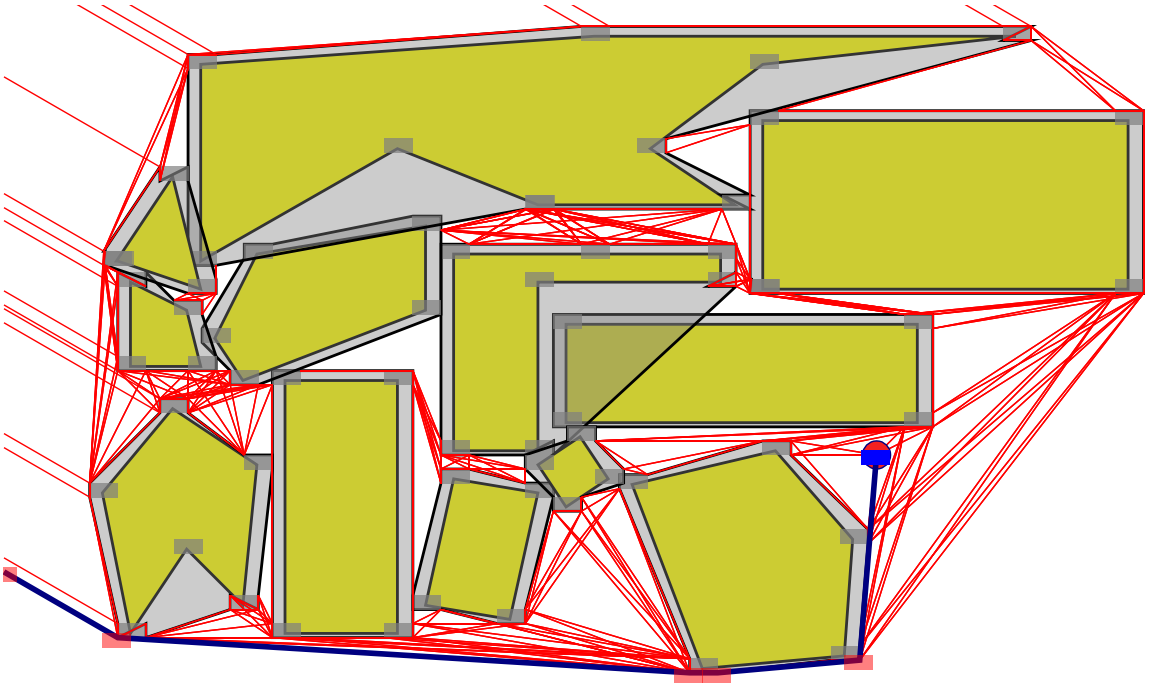
Ausgabe:

```
Vis. Graph time: 0.1669835358079239
Dijkstra   time: 0.0015781862629355992

Startzeit: 7:27:54 Uhr (abgerundet)
Zielzeit:  7:30:40 Uhr (abgerundet)
y-Koord:   340
Wegdauer:  2 Minuten 45 Sekunden (abgerundet)
Weglänge:  691m
WEG:
Starte beim Haus (621|162) (ID: 'L')
gehe zu (410|175) (ID: 'P8')
gehe zu (380|165) (ID: 'P3')
gehe zu (320|165) (ID: 'P3')
gehe zu (280|215) (ID: 'P5')
gehe zu (170|215) (ID: 'P6')
gehe zu (130|265) (ID: 'P9')
gehe zu (  0|340) (ID: keine (Endpunkt))
```

Hier gibt es abermals vergleichsweise „viele“ Polygone, von denen aber auch immerhin fast die Hälfte benötigt wird.

Erweiterung der Aufgabe

Hier soll statt eines Punktes ein Rechteck zur y-Achse gelangen:



Auch hier funktioniert die Vergrößerung der Polygone meistens korrekt. Das Rechteck ist jedoch zu groß für die meisten Lücken und deshalb wird ein Weg am Rand entlang gewählt.

Quellcode

Bedeutung der Ordner und Skripte

```
Aufgabe1/beispieldaten/
```

Enthält die Beispieldaten der BwInf-Webseite. Außerdem sind die Dateien, die für die Erweiterung der Aufgabe in den Beispielen verwendet wurden, enthalten. Diese sind mit `-zusatz.txt` am Ende des Dateinamens gekennzeichnet.

```
Aufgabe1/beispieldaten-svg/
```

Enthält die SVG-Dateien der Beispieldaten.

```
Aufgabe1/solutions/
```

Enthält die SVG-Dateien aus `beispieldaten-svg`, denen jeweils der beste Weg eingefügt wurde.

```
Aufgabe1/site-packages/
```

Enthält das `svgwrite`-Modul.

```
python3 Aufgabe1/aufgabe1.py
```

Berechnet die Beispieldaten und gibt die geforderten Ausgaben aus. Wenn der Ordner `beispieldaten-svg` mit entsprechenden Dateien existiert, werden die berechneten Wege als SVG-Grafiken im Ordner `solutions` gespeichert.

`Aufgabe1/way_searcher.py`

Modul, das die `WaySearcher`-Klasse enthält. Wenn diese Datei mit `python3` ausgeführt wird, wird der Weg von einer (Beispiel)Daten-Datei (erster Parameter) berechnet und in einer SVG-Grafik (zweiter Parameter) gespeichert.

`Aufgabe1/geometry.py`

Enthält alle Klassen, um Punkte, Nodes, Kanten, Strecken usw. darzustellen und Schnittpunkte zu berechnen.

Modul geometry

Hier findet sich der Inhalt des Moduls `geometry`. Quellcode-Ausschnitte werden vor allem im nächsten Abschnitt, in dem das Modul `way_searcher` behandelt wird, gezeigt, denn das `geometry`-Modul beinhaltet viele kleine Funktionen, bei denen auch eine kurze Beschreibung ausreicht.

Klasse Point

Diese Klasse stellt einen Punkt im Koordinatensystem mit den zwei Attributen `x` und `y` dar, die dem Konstruktor entsprechend übergeben werden.

`def get_angle(other)`

Diese Methode berechnet den Winkel zu einem anderen Punkt, der mit dem Parameter `other` angegeben wird. `get_angle` wird nicht dazu verwendet, die Winkel zwischen den Punkten beim Erstellen des Sichtbarkeitsgraphen zu berechnen (dazu siehe `LineSegment`), sondern, um die Punkte bei der Vergrößerung eines Polygons (siehe die Erweiterung der Aufgabe) zu sortieren. Benutzt wird die `math.atan2`-Funktion wie beim `LineSegment`.

`def get_distance(other)`

Gibt die Distanz zum Punkt `other` zurück. Wird u.a. vom `LineSegment` benutzt, um die Länge zu berechnen.

`def has_same_point(other)`

Vergleicht die Koordinaten dieses Punktes und von `other` auf Gleichheit. Dadurch wird nicht (wie mit `==`) überprüft, ob es sich um dasselbe Objekt handelt. Das Überschreiben der `__eq__`-Methode ist keine Alternative, denn die Möglichkeit, mit `==` auf gleiche Objekte zu überprüfen, wird auch benötigt.

Klasse Node

Diese Klasse erbt von Point und stellt einen Knoten dar. Als Parameter bekommt der Konstruktor wahlweise auch die Kanten, von denen er ein Eckpunkt ist, und die Nachbarn übergeben, die jeweils in den Attributen `edges` und `neighbors` gespeichert werden. Außerdem gibt es die Attribute `last_time` (letztmöglicher Zeitpunkt für Dijkstra-Algorithmus), `parent` (auch Dijkstra-Algorithmus), `polygon` (das Polygon-Objekt, zu dem dieser `Node` gehört, oder `None`) und `polygon_id` (die ID des Polygons, zu dem der `Node` gehört). Das Attribut `polygon_neighbors` speichert die zwei Knoten, mit denen dieser Node durch eine Kante verbunden ist.

`def get_rotated_on_y_axis(angle=30)`

Gibt einen `Node` zurück, der sich `angle`° über diesem `Node` befindet. Benutzt wird die `math.tan`-Funktion.

`def reload_last_time(new_parent, speed)`

Berechnet den spätmöglichsten Zeitpunkt mit einem neuen Parent `new_parent` und der Geschwindigkeit `speed` (normalerweise Lisas Geschwindigkeit) neu. Ist das Ergebnis besser (d.h. größer) als `self.last_time`, werden `new_parent` und der neu berechnete letztmögliche Zeitpunkt übernommen.

Klasse LineSegment

Diese Klasse stellt eine Strecke dar, die als Parameter zwei Punkte bekommt, von denen der erste im Attribut `p1` und der zweite im Attribut `p2` gespeichert werden. Berechnet wird außerdem die Länge mit `Point.get_distance` im Attribut `length` und der Winkel im Attribut `angle`, wobei der erste Punkt als Ursprung des Winkels benutzt wird. `angle` wird zum Erstellen des Sichtbarkeitsgraphen benötigt und ist dementsprechend 0, wenn die Strecke nach unten zeigt.

`def in_range(point)`

Diese Methode wird von `get_intersection` (sucht einen Schnittpunkt zwischen zwei Strecken) aufgerufen. Da in `get_intersection` zunächst von zwei Geraden ausgegangen wird, muss noch geprüft werden, ob sich der entstandene Schnittpunkt tatsächlich innerhalb der Strecken befindet. Das übernimmt diese Methode, die dementsprechend `True` oder `False` zurückgibt.

Klasse Edge

Diese Klasse erbt von `LineSegment` und funktioniert auch fast genauso. Allerdings werden im Konstruktor nur `Nodes` zugelassen, denn diese Klasse steht für eine Kante eines Polygons. Den beiden Knoten wird jeweils das neue `Edge`-Objekt im Konstruktor als Kante zum Attribut `Node.edges` hinzugefügt und beide Punkte erhalten den jeweils anderen Punkt zu ihrem Attribut `polygon_neighbors` hinzu.

Klasse Polygon

Die Klasse stellt ein Polygon dar und bekommt eine Liste aller Eckpunkte, die Polygon-ID und optional eine weitere Liste von Punkten übergeben. Die zweite Liste von Punkten enthält alle Verschiebungen, um das Polygon zu vergrößern (siehe die Erweiterung der Aufgabe), denn Lisas Polygon wird relativ zum Fixpunkt (Lisas Haus) gespeichert. Der Quellcode für die Vergrößerung ist folgendermaßen aufgebaut:

```
def __init__(self, vertices: List[Node], polygon_id=None, addition: List[Point] = None):
    […]  # Erstelle Polygon wie gewöhnlich
    if addition:  # nur, wenn ein Polygon für die Vergrößerung angegeben wurde
        new_vertices = []  # alle neuen Punkte (in den Bildern kleine graue Polygone)
        # Füge für jeden Punkt das Polygon in 'addition' hinzu (180°-Drehung durch
        # Subtraktion)
        for vertex in vertices:
            addition_points = [Node(vertex.x - point.x, vertex.y - point.y)
                                                                for point in addition]
            # nur Punkte, die nicht im Polygon liegen
            new_vertices.extend(point for point in addition_points
                                                    if not self.point_in_polygon(point))
        midpoint = self.calc_midpoint()  # Mittelpunkt ist Durchschnitt aller Punkte
        # Sortiere nach Winkel
        vertices = sorted(new_vertices, key=lambda p: midpoint.get_angle(p))
        # Erstelle alle Kanten zu den Punkten
        new_edges = [LineSegment(vertices[i], vertices[i - 1])
                                                        for i in range(len(vertices))]
        first_overlapping = True  # Schneidet die erste Kante das originale Polygon?
```

```
        for edge in self.edges:
            if get_intersection(new_edges[0], edge):
                break
        else:
            first_overlapping = False
        last_overlapping = first_overlapping  # Schneidet die zuletzt überprüfte Kante
                                              # das originale Polygon?
        for i in range(len(vertices) - 1, -1, -1):  # rückwärts, damit Löschen
                                                    # funktioniert und der Index sich nicht verschiebt
            for edge in self.edges:
                if get_intersection(new_edges[i], edge):
                    break
            else:
                # wird nur ausgeführt, wenn die Schleife nicht mit 'break' abgebrochen
                # wurde:
                last_overlapping = False  # kein Schnittpunkt, Kante ist nicht im Polygon
                continue
            if last_overlapping:
                # Sowohl diese als auch die letzte Kante schneiden das originale Polygon,
                # also:
                # Lösche den zugehörigen Punkt
                del vertices[i]
            last_overlapping = True
        # Überprüfe, ob die erste und die letzte Kante das originale Polygon schneiden
        if last_overlapping and first_overlapping:
            del vertices[0]
        # Berechne die Kanten mit den gefilterten Punkten neu
        # der folgende Teil wird auch oben für ein gewöhnliches Polygon ausgeführt
        # der Teil in '[…]' muss trotzdem ausgeführt werden, damit das originale Polygon
        # erstellt wird und mit .point_in_polygon() geprüft werden kann, ob sich ein Punkt
        # im Polygon befindet, außerdem werden die originalen Kanten benötigt
        self.edges = [Edge(vertices[i], vertices[i - 1]) for i in range(len(vertices))]
        self.points = vertices
        for node in self.points:
            node.polygon = self
            node.polygon_id = polygon_id
```

```
@staticmethod
def from_str(text, polygon_id, addition)
```

Erstellt ein neues Polygon aus einer Zeile (`text`) mit `polygon_id` und optional einer Liste von Punkten, die die Vergrößerung angibt (normalerweise `WaySearcher.lisa_polygon`).

```
def calc_midpoint()
```

Berechnet den Mittelpunkt des Polygons mit dem Durchschnitt aller beteiligten Knoten.

```
def point_in_polygon(point)
```

Prüft, ob `point` in diesem Polygon liegt, indem jeder `edge` in `self.edges` durchgegangen wird und alle Schnittpunkte gezählt werden (siehe Lösungsidee).

Funktion `get_intersection(line1, line2)`

Es wird der Schnittpunkt zwischen den `LineSegment`-Objekten `line1` und `line2` berechnet und als Point zurückgegeben, oder – wenn es keinen Schnittpunkt gibt – `None`. Der Schnittpunkt wird zunächst mit dem Kreuzprodukt so berechnet, als ob es sich um zwei Geraden handeln würde, und dann wird überprüft, ob sich dieser Schnittpunkt überhaupt auf den zwei Strecken befinden kann, indem für jede Strecke die Methode `.in_range()` aufgerufen wird.

Modul way_searcher

Klasse WaySearcher

`def __init__(polygons, lisa_node, bus_speed=30/3.6, lisa_speed=15/3.6)`
Initialisiert einen `WaySearcher` mit den übergebenen Parametern. `polygons` ist eine Liste von `Polygon`-Objekten, die sich im Koordinatensystem befinden. Aus diesen Polygonen werden alle `Nodes` und `Edges` für die Attribute `nodes` und `edges` gesucht. Der Liste `nodes` wird auch `lisa_node` – Lisas Haus – hinzugefügt, aber für Lisas Haus gibt es auch das separate Attribut `lisa_node`.

`def from_str(text)`
Liest `text` – einen String – gemäß dem auf der BwInf-Webseite beschriebenen Format ein und gibt einen initialisierten `WaySearcher` zurück.

`def create_visibility_graph(check_overlapping_polygons)`
Erstellt einen Sichtbarkeitsgraphen, indem alle Nodes in `WaySearcher.nodes` ihre jeweiligen `neighbors` zugewiesen bekommen. Nur wenn der einzige Parameter True ist, werden alle Punkte, die sich innerhalb eines Polygons befinden, ausgeschlossen, weil dies länger dauert. Hier ist der (im Gegensatz zur Originaldatei auf Deutsch) kommentierte Quellcode:

```
# der nodes-Liste werden später noch Nodes auf der y-Achse hinzugefügt, deshalb wird eine
# Kopie gespeichert
all_nodes = self.nodes.copy()
if check_overlapping_polygons:
    # Suche alle Punkte, die sich nicht in einem Polygon befinden oder Lisas Haus sind
    nodes_outside_polygons = [point for point in all_nodes
                              if (point == self.lisa_node or  # nicht Lisas Haus
                                  not any(polygon.point_in_polygon(point)
                                          for polygon in self.polygons
                                      if polygon != point.polygon)  # in keinem Polygon
                                  )]
else:
    nodes_outside_polygons = all_nodes

# Berechne den Winkel, um den die y-Nodes höher sind (normalerweise 30°)
y_rotation_angle = math.degrees(math.asin(self.lisa_speed/self.bus_speed))
for origin in nodes_outside_polygons:  # 'origin' ist der Ursprungspunkt
    # Berechne zugehörigen Punkt auf der y-Achse
    y_node = origin.get_rotated_on_y_axis(y_rotation_angle)
    # Erstelle Strecken für jeden anderen Node. Für jedes LineSegment wird in der
    # __init__-Methode der Winkel berechnet (segment.angle). Sollten Winkel gleich sein,
    # wird nach der Entfernung des Punkts (segment.length) sortiert.
    lines = sorted((LineSegment(origin, point) for point in all_nodes + [y_node]
                                                        if point != origin),
                   key=lambda segment: (segment.angle, segment.length))

    # die Kanten-Liste: alle Kanten, die für die Sichtbarkeit relevant sein können:
    test_edges = set()
    # Fülle test_edges mit Kanten, die 'down' (Linie vom Ursprungspunkt nach unten)
    # schneiden
    down = LineSegment(origin, Point(origin.x, 0))  # angle = 0° (start)
    for edge in self.edges:
        intersection = get_intersection(down, edge)
        # es muss einen Schnittpunkt geben (nicht None), und wenn der Schnittpunkt gleich
        # einem Endpunkt (edge.p1 und edge.p2) ist, muss der jeweils andere Punkt links
        # liegen
        # .has_same_point() überprüft, ob die Koordinaten (x/y) gleich sind, nicht, ob die
        # Objekte gleich sind (also zum Vergleichen von Koordinaten geeignet)
```

```
        if intersection is not None and \
          ((not intersection.has_same_point(edge.p1) or intersection.x >= edge.p2.x) and
           (not intersection.has_same_point(edge.p2) or intersection.x >= edge.p1.x)):
            test_edges.add(edge)
    for line in lines:  # gehe alle Strecken zu den Punkten durch
        # line.p1 ist der Ursprungspunkt, line.p2 der Punkt, der getestet wird
        if line.p2 in nodes_outside_polygons or line.p2.x == 0:  # der Punkt darf nicht
                                                 # ausgeschlossen werden oder muss auf
                                                 # der y-Achse liegen (also y_node sein)
            for edge in test_edges:
                if get_intersection(edge, line) is not None:
                    break  # 'line' schneidet sich mit 'edge'
            else:
                # der 'else'-Teil nach einer Schleife wird nur aufgerufen, wenn sie nicht
                # mit break abgebrochen wurde.
                # hier wurde also keine Überschneidung festgestellt
                # Die folgende if-Bedingung enthält die drei Bedingungen aus der
                # Umsetzung: unterschiedliche Polygone bzw. keine Polygone zugeordnet,
                # direkte Nachbarn im Polygon, Mitte der Strecke außerhalb des Polygons
                if line.p1.polygon_id != line.p2.polygon_id or \
                    line.p1.polygon is None or line.p2.polygon is None or \
                    line.p2 in line.p1.polygon_neighbors or \
                    not line.p1.polygon.point_in_polygon(
                       Point((line.p1.x + line.p2.x) / 2, (line.p1.y + line.p2.y) / 2)):
                    # 'line.p2' ist sichtbar, also wird 'line.p1' als Nachbar hinzugefügt
                    # (nur in eine Richtung, in andere Richtung, wenn line.p2
                    # Ursprungspunkt ist)
                    line.p2.neighbors.append(line.p1)
                    self.vis_graph_lines.add(line)  # für die Darstellung als svg
        # Aktualisiere die Kanten-Liste mit der symmetrischen Differenz (^)
        test_edges ^= line.p2.edges
    if y_node.neighbors:
        # nur, wenn der y_node einen Nachbarn (nämlich 'origin') hat, ist er von 'origin'
        # aus sichtbar und wird hinzugefügt. Auch der späteste  Zeitpunkt, zu dem dieser
        # Punkt erreicht werden muss, wird berechnet.
        y_node.last_time = y_node.y / self.bus_speed
        self.nodes.append(y_node)
self.created_vis_graph = True  # Attribut wird später vom Dijkstra-Algorithmus überprüft
                               # und ist ansonsten auf 'False' gesetzt
```

`def dijkstra()`

Diese Methode berechnet den besten Weg mit dem Dijkstra-Algorithmus. Das `last_time`-Attribut der Knoten wurde bereits im Konstruktor auf `-math.inf` gesetzt und für die Knoten auf der y-Achse die entsprechende Zeit des Busses berechnet. Der Quellcode sieht folgendermaßen aus:

```
if not self.created_vis_graph:  # sicherstellen, dass der Sichtbarkeitsgraph bereits
                                # erstellt wurde
    raise ValueError("Visibility graph must be generated first. ")
unvisited_nodes = self.nodes.copy()  # Liste, die alle noch nicht besuchten Knoten enthält

while unvisited_nodes:  # solange Knoten noch nicht besucht wurden
    # Wähle Knoten mit größter letztmöglicher Zeit
    next_node = max(unvisited_nodes, key=lambda n: n.last_time)
    if next_node == self.lisa_node:
        # Lisas Haus wurde gefunden, der beste Weg wurde gefunden
        break
    unvisited_nodes.remove(next_node)  # 'next_node' wird jetzt "besucht"
    # Berechne Zeit für jeden Nachbarn neu
    for neighbor in next_node.neighbors:
        if neighbor in unvisited_nodes:  # Knoten, die bereits besucht wurden, haben immer
                                         # schon die beste Zeit gespeichert und werden
```

```
                                        # nicht nochmal aktualisiert
            # siehe Dokumentation der Node.reload_last_time-Methode
            neighbor.reload_last_time(next_node, self.lisa_speed)
else:
    # die Schleife wurde nicht mit 'break' abgebrochen, also konnte Lisas Haus nicht
    # erreicht werden
    raise ValueError("Lisa's house ('{}') is not reachable".format(self.lisa_node))
# Lisas Haus wurde gefunden
# Berechne den Weg mit dem 'parent'-Attribut der Nodes
way_points = []
node = self.lisa_node  # starte bei Lisas Haus
length = 0  # Länge des Wegs in Metern
while node is not None:
    way_points.append(node)
    if node.parent:
        # node.parent ist nicht 'None'
        length += node.get_distance(node.parent)
    node = node.parent
    # sollte 'node.parent' 'None' sein (der Knoten liegt also auf der y-Achse), wird jetzt
    # durch die while-Schleife abgebrochen
return way_points, length, length/self.lisa_speed  # alle Eckpunkte des Weges, die Länge,
                                                    # Lisas benötigte Zeit
```