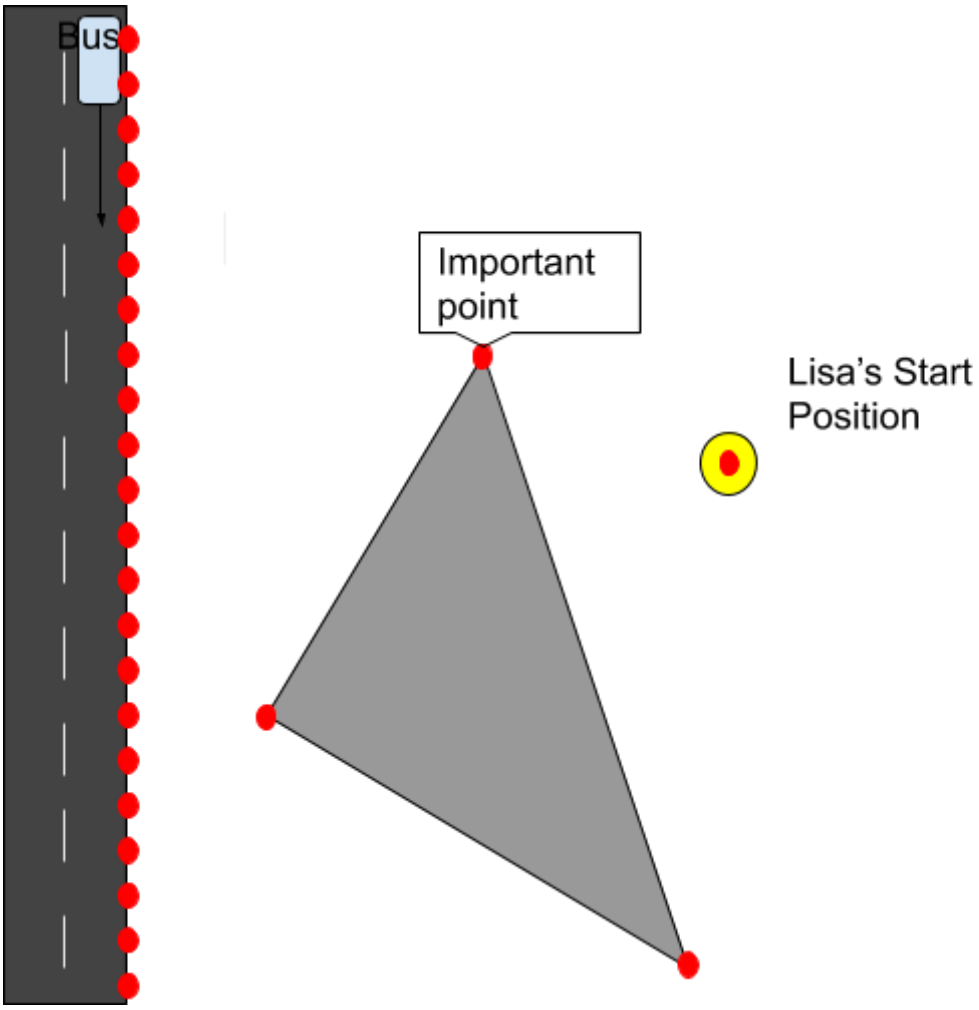
Aufgabe1: “Lisa rennt”

Inhaltsverzeichnis

Grundidee:

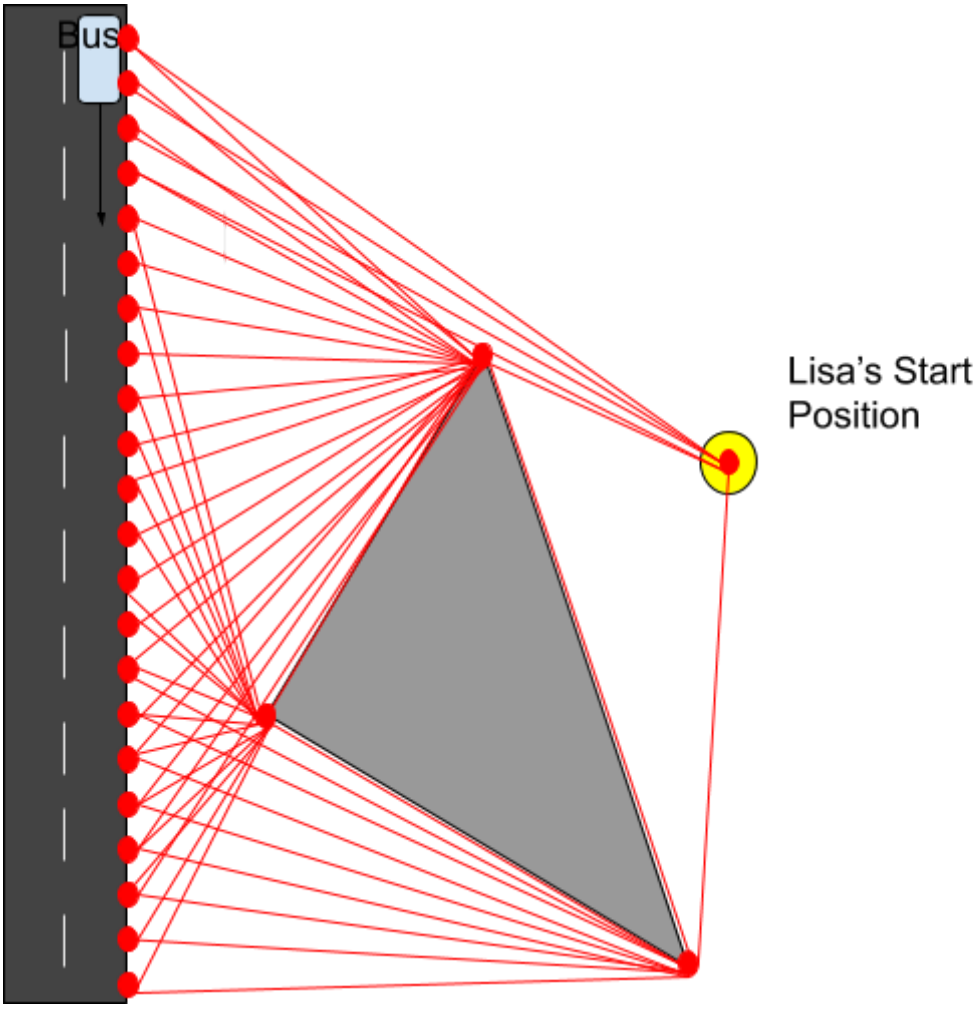
Es ergib nur Sinn für Lisa zu Ecken von den Hindernissen zu laufen oder zu Punkten an den Straßen. Lisa ist schneller, wenn sie von Ecke zu Ecke läuft. Punkte, die keine Ecken sind und nicht an der Straße sind, können komplett ignoriert werden, Lisa kann nie schneller sein, wenn sie zu Punkten läuft, die keine Eckpunkte sind.



In dieser Zeichnung sind alle wichtigen Punkte eingezeichnet. An der Strasse sind der Übersicht wegen nur einige Punkte eingezeichnet worden, eigentlich gäbe es viel mehr Punkte. So viele Punkte wie viele Meter die Straße lang ist.

Erste Lösungsidee:

Es werden alle Linien von allen wichtigen Punkten zu allen wichtigen Punkten durchgegangen, es wird bei jeder Linie getestet, ob sie ein Polygon kreuzt. Lisa kann an Gebäuden entlang laufen. Unten sieht man alle Linien, die nicht von Polygonen gekreuzt werden.



Für uns ist dies sehr anschaulich, aber der Computer bevorzugt eine andere Darstellung. Die Datenstruktur Graph.
Deshalb müssen wir einen Graphen erstellen, mit dem wir dann auch den kürzesten Weg berechnen werden, aber dazu komme ich noch.

Umsetzung:

Ein **Graph** in Informatik ist etwas ganz anderes als ein Graph in Mathe. Ein Graph ist sehr simpel, er besteht nur aus zwei Sachen “vertices” (Knotenpunkte) und “edges” (Kanten).

Kanten:

- verbinden zwei Knotenpunkte miteinander.
- speichern Kosten (in unserem Fall die Entfernung in Metern von einem Punkt zu einem anderen)

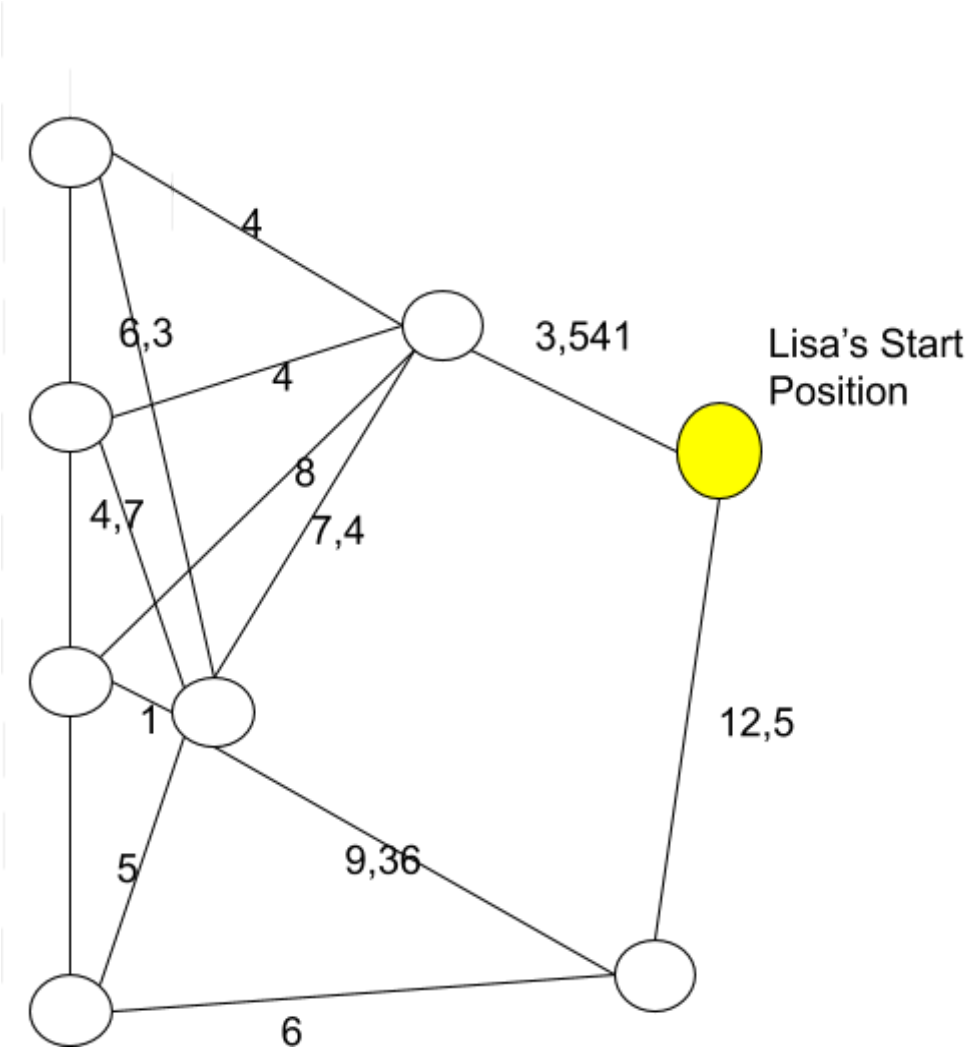
Knotenpunkte:

- haben einen Namen, mit dem man sie identifizieren kann
- haben Kanten, mit denen sie zu anderen Knotenpunkten verbunden sind

Graph:

- hat Knoten
- hat Kanten

Also erstellen wir einen Graphen. Jeder wichtige Punkt wird als Knotenpunkt zu unserem Graphen hinzugefügt. Jede Linie, die keine Polygone kreuzt, ist eine Kante in unserem Graph, die den Abstand von den beiden Punkten als Kosten speichert.



weiße Kreise sind Knotenpunkte
Linien sind Kanten
Zahlen neben den Kanten sind die Kosten.

Um jetzt den kürzesten Weg vom Startpunkt zu allen Punkten zu finden, verwende ich den Djikstra Algorithmus.

Nun sind wir schon sehr weit, wir haben zu allen Punkten den kürzesten Weg.
Aber wir versuchen ja den schnellsten Weg zu finden.

Also werden die kürzesten Wege von der Startposition zu allen Straßenpunkten miteinander verglichen.
Es wird ermittelt, wie lange Lisa braucht zu dem Punkt, minus, wie lange der Bus braucht bis zu diesem Punkt.

So wird der beste Weg herausgefunden.
Super! Aufgabe gelöst!

Warum der **Djikstra** Algorithmus ?

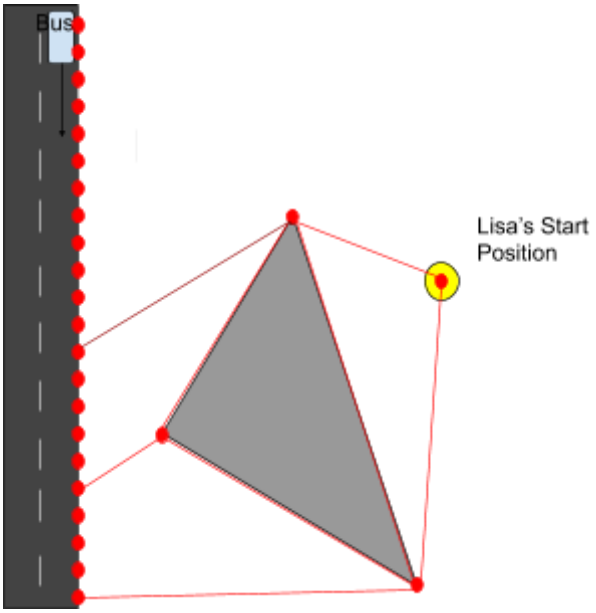
Djikstra Algorithmus findet den schnellsten Weg zu allen Knotenpunkten. Es gibt, aber noch einen weiteren Algorithmus den A * Algorithmus gesprochen A star auf Deutsch A stern,der auch den schnellsten Weg in einem Graphen findet, dieser ist sogar schneller als der Djikstra Algorithmus, er funktioniert fast genauso wie dieser, er ist quasi eine Erweiterung des Djikstra Algorithmus. Es wird zielgerichteter gesucht,mit Hilfe einer Schätzfunktion. Da wir aber den kürzesten Weg zu mehreren Punkten berechnen müssen, funktioniert der A stern Algorithmus nicht.

Optimierungen:

Diese erste Lösung findet immer den richtigen Weg, funktioniert einwandfrei, es trat nur ein Problem auf: Der Computer brauchte sehr lang, um den besten Weg zu berechnen. Deswegen habe ich überlegt, wie man den Prozess verbessern kann.

Optimierung durch sparen an Straßnpunkten

Man braucht keine Verbindung zu allen Straßenpunkten. Der optimale Winkel ist 30 Grad. Eine Ecke von einem Hindernis wird nicht mit allen Straßenpunkten verbunden, sondern nur mit dem Straßenpunkt bei 30 Grad.



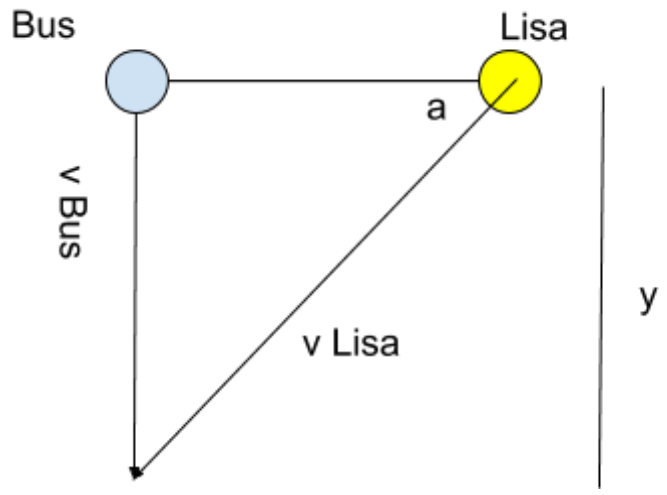
So verringern wir die Anzahl an Kanten und Knotenpunkte im Graphen -> Dadurch braucht der Djikstra Algorithmus nicht mehr so lange.

Es muss nicht so oft getestet werden, ob eine Linie Polygone schneidet. Wo vorher 1000 mal getestet wurde (wenn das Fenster 1000 Pixel Vertical hat- das entspricht einer Straßenlänge von 1000 m), brauchen wir jetzt nur noch einmal zu testen. Anstatt T(n) = 1000 * n (n Anzahl der Polygon Eckpunkte) nur noch T(n) = n
Also **sehr** viel schneller.

Warum sind 30 ° der optimale Winkel?

Wir gehen davon aus, dass Lisa und der Bus auf der gleichen Höhe starten. Nun wollen wir herausfinden, in welchem Winkel Lisa am Besten laufen sollte.

gegeben: vBus(Geschwindigkeit des Buses) = 30km/h; vLisa (Geschwindigkeit von Lisa) = 15 km/h; y(y Differenz in m von Lisa und Bus zum Ziel)
gesucht: a (Winkel siehe Zeichnung)



Jetzt brauchen wir die Physik-Formel für die Strecke .

s = v * t (gleichförmige Bewegung, d.h. keine Beschleunigung)

In unserem Fall ist s = y wir betrachten die Strecke y.

y = vBus * tBus
Nun müssen wir auf die Zeit umstellen.

tBus = y / vBus

Perfekt !!

Jetzt noch das Gleiche für Lisa:

Erst müssen wir berechnen, wie lang der Weg ist, den Lisa laufen muss. Dieser Weg ist davon abhängig, wie groß der Winkel a ist und wie lange y ist .
Mit simpler Trigonometrie: sin(a) * y = sLisa (die Strecke, die Lisa laufen muss)

Jetzt die Gleichung abhängig zur Zeit:

tLisa = sin(a) * y / vLisa

Jetzt gehen wir davon aus, dass tLisa = tBus, da wir wollen, dass Bus und Lisa zur gleichen Zeit ankommen, sodass Lisa in den Bus einsteigen kann.

Jetzt können wir beide Gleichungen gleichsetzten .

y / vBus	= sin(a) * y / vLisa	| / y
1 / vBus	= sin(a) / vLisa	| * vLisa
vLisa / vBus	= sin(a)	| hoch -1
a	= sin^-1(vLisa / vBus)	

Jetzt einsetzen von 15 für vLisa und 30 für vBus

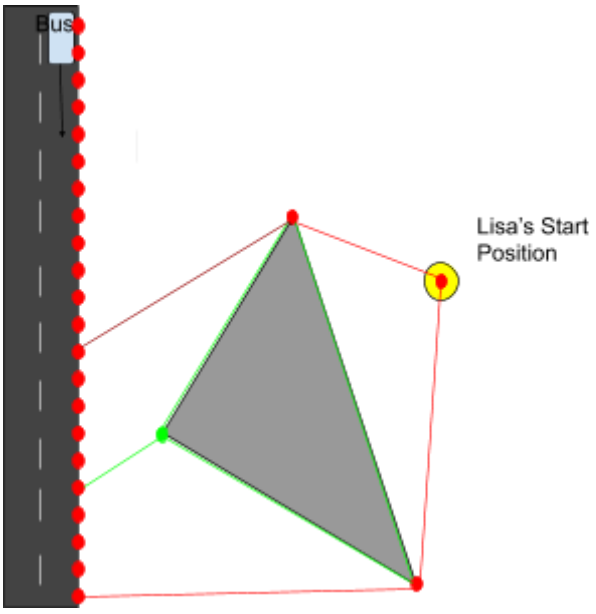
a = sin ^-1(15/30)
a = sin ^-1(1/2)
a = 30°

Optimierung durch Verwenden einer priority Queue:

Djikstra wird mit einer priority Queue verwendet, implementiert als Heap.
So bekommt man aus Djikstra **O**(**V * V**) ; **O**(**E + V log V**) in der Big O Notation. **V** sind **v**ertices(Knotenpunkte) und **E** ist die Anzahl der **E**dges (Kanten). Also deutlich schneller.

weitere Optimierung:

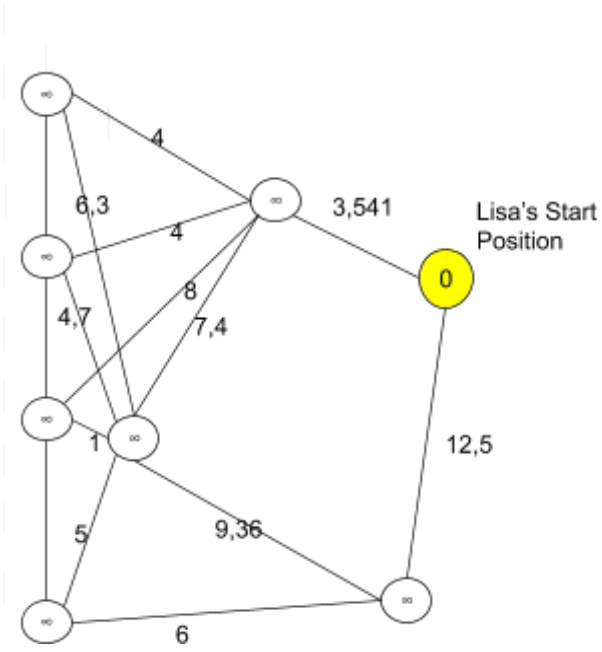
Es müssen keine Verbindungen von Kanten, von denen der 30 ° Weg frei ist zu anderen Kanten bei denen der 30° Weg frei ist, erstellt werden. Wenn der 30 ° Weg frei ist ,wissen wir, dass kein Weg besser sein kann als der 30 ° Weg ist. Es macht nie Sinn von einem Punkt, bei dem der 30 ° Weg frei ist, zu einem Punkt zu gehen, bei dem der 30° Weg frei ist.



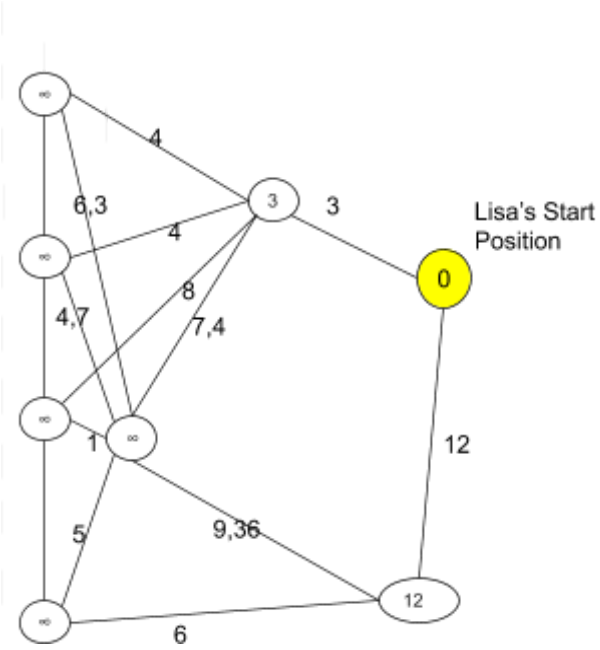
grüne Linien: Kanten, welche wir uns jetzt sparen können.
rote Linien: Kanten, welche immer noch gebraucht werden.

Wie **funktioniert** der **Djikstra Algorithmus** :

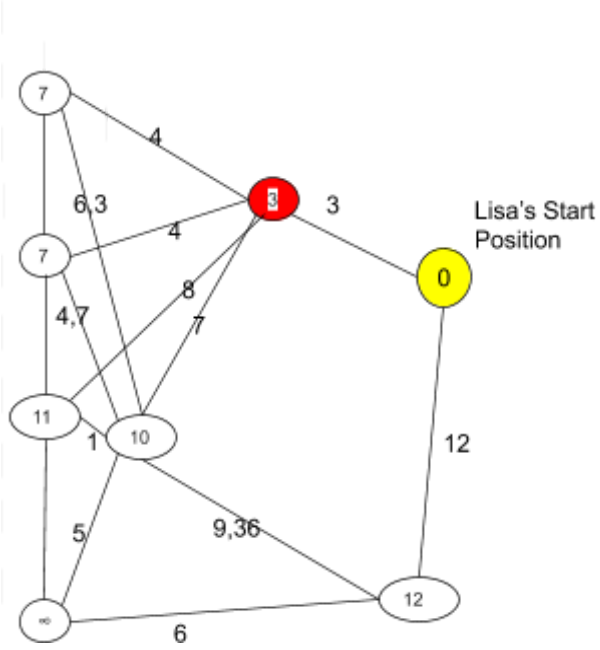
Zuerst werden alle Werte der Nodes auf unendlich gesetzt. Dafür wird ein Dictionary verwendet, also Key-Value Paare.
Nur der Start-Knotenpunkt, Lisas Haus, wird auf null gesetzt. Lisa braucht von Lisa's Haus zu Lisa's Haus 0 Meter zu laufen.



Jetzt werden zuerst alle verbundenen Knotenpunkte auf den Wert des jetzigen Knotenpunktes plus die Kosten der Kante gesetzt, wenn der neue Wert kleiner ist als der Alte.



Nun wird mit dem kleinsten Knotenpunkt der Prozess wiederholt. Also hier 3: Alle Knotenpunkte, die mit 3 verbunden sind, werden auf 3 + die Kosten der Kante gesetzt, wenn der neue Wert kleiner ist als der alte.



Dieser Prozess wird für alle Knoten wiederholt.

Nun, der Quellcode:

```
def _djikstra(self, from_node: Node):
    assert from_node in self._nodes.values(), "Node has to be in the
Graph!"
    print(f"Djikstra Algorithm on {len(self._nodes)} Nodes and
{len(self._edges)} Edges")
    seen: Set[Node] = set([])
    nodes_heap = []
    distances: Dict[Node, int] = {}
    for node in self._nodes.values():
        if node is from_node:
            distances[node] = 0
            heap_entry = HeapEntry(0, node)
            heapq.heappush(nodes_heap, heap_entry)
        else:
            distances[node] = sys.maxsize
            heap_entry = HeapEntry(sys.maxsize, node)
            nodes_heap.append(heap_entry)
    came_from: Dict[Node, Node] = {}  # speichert zu einer Node immer
von welcher parent Node man am schnellsten zu dieser Node kommt,so
lässt sich der Weg rekonstruiren

    while len(nodes_heap) > 0:
        current_node: Node = heapq.heappop(nodes_heap).node
        current_node_cost: int = distances[current_node]
        seen.add(current_node)
        for edge in current_node.edges:
            connected_node: Node = edge.get_other_node(current_node)
            if connected_node in seen:
                continue
            connected_node_cost: int = current_node_cost + edge.cost
            if connected_node_cost < distances[connected_node]:
                distances[connected_node] = connected_node_cost
                heapq.heappush(nodes_heap,
                    HeapEntry(connected_node_cost, connected_node))
                came_from[connected_node] = current_node
    return came_from, distances
```

Diese Methode ist **privat** (dies zeigt der Unterstrich in python **_djikstra**), da diese Methode nur innerhalb der Klasse aufgerufen werden soll.
Von außerhalb der Klasse soll die **public** Methode **shortest_path**() aufgerufen werden.

```
def shortest_path(self, from_node_position: List[int],
to_node_position: List[int]):
    assert str(from_node_position) in self._nodes.keys() and
str(to_node_position) in self._nodes.keys(), "Nodes have to be in
the graph !"
    from_node: Node = self._nodes[str(from_node_position)]
    to_node: Node = self._nodes[str(to_node_position)]
    if self._came_from is None:
        self._came_from, self._distances =
self._djikstra(from_node)
    return _reconstruct_path(self._came_from, from_node, to_node),
self._distances[to_node]
```

Die Methode **shortest_path** ruft dann intern die Methode djikstra auf,wenn djikstra noch nicht aufgerufen wurde. Wenn man den Pfad zu mehreren Knotenunkte wissen will (was wir auf jeden Fall wollen, nämlich alle Straßenpunkte), wird also nur einmal Djikstra aufgerufen. Die Methode **_reconstruct_path()** rekonstruiert den Pfad, mit Hilfe des came_from dictionaries (in anderen Sprachen Map genannt (besteht aus Key,Value Paaren). **_reconstruct_path** ist auch privat (_) , da sie nicht von außerhalb der Klasse aufgerufen werden muss, bzw. soll.

```
def _reconstruct_path(came_from: Dict[Node, Node], from_node:
Node, to_node: Node):
    path: List[Node] = [to_node]  # could be a set
    while path[0] is not from_node:
        path.insert(0, came_from[path[0]])
    return path
```

Was ist eine **priority Queue**?

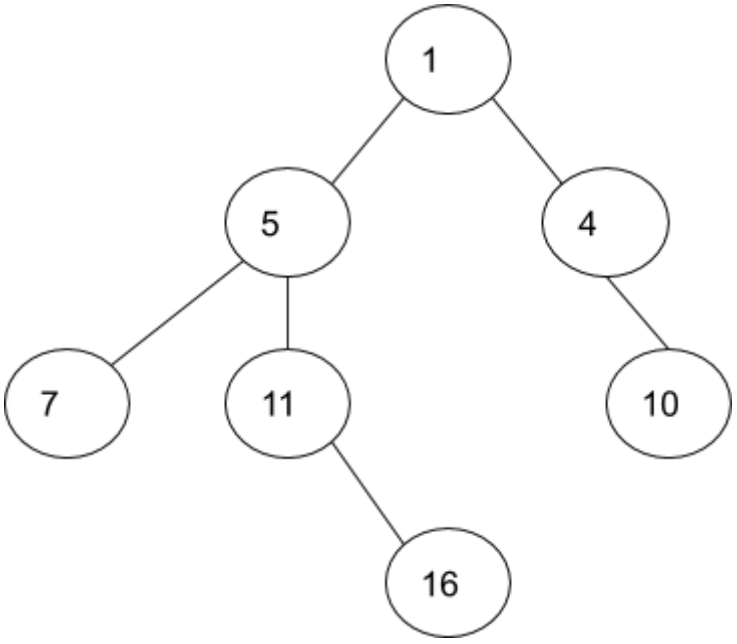
Eine **priority Queue** ist ein abstrakter Datentyp. Priority Queue beschreibt ein Prinzip, das jedes Element in der priority Queue eine **Priorität** (priority) hat.
Eine priority Queue muss:

- Eine Methode haben, mit der man das Element mit der niedrigsten Priorität abruft und entfernt. Meistens pop genannt.
- Eine Methode haben, mit der man ein Element mit priority hinzufügen kann. Meistens push genannt.
- testen ob die Priority Queue leer ist.

Eine **Priority Queue** ist ein abstrakter Datentyp, dies bedeutet, dass sie durch verschiedene Datentypen implementiert werden kann. Zum Beispiel durch eine Linked List, bei der man um das kleinste Element zu finden einfach alle miteinander vergleicht.
Die schnellste Implementierung einer Priority Queue ist durch ein **Heap**. Deswegen habe ich ein Heap verwendet.

Was ist ein **Heap** ?

Ein **Heap** ist eine Art eines Trees (Baum), in der eine **Child** Node immer **größer** ist als die **Parent** Node. In einer **max-heap** ist es genau andersrum: Child Node ist immer **geringer** als die Parent Node.



Es gibt verschiedene Arten von Heaps:
Eine davon ist der **binärer Heap**.
Das Element mit der kleinsten Priorität abzurufen und zu entfernen: **pop** hat **O**(**log n**)
Ein Element hinzuzufügen: **push** hat O(**log n**).

Umsetzung:

Für die Umsetzung habe ich die Programmiersprache **Python** verwendet, da sie sich sehr für die Aufgabe eignet. Da Python eine **höhere Programmiersprache** ist, kann man sehr schnell mit ihr entwickeln und schnell erste Ergebnisse sehen. Man muss Variablen zum Beispiel keinem Datentyp zuordnen, bei Funktionen kein Rückgabewert angeben uvm. Sachen, die das Entwickeln verschnellern. Man muss insgesamt viel weniger Zeilen Code schreiben, man ist viel schneller, als wenn man eine niedrigere (niedriger als Python aber

trotzdem eine hohe Programmiersprache) Programmiersprache wie Java,C,C++,C# etc. verwendet.

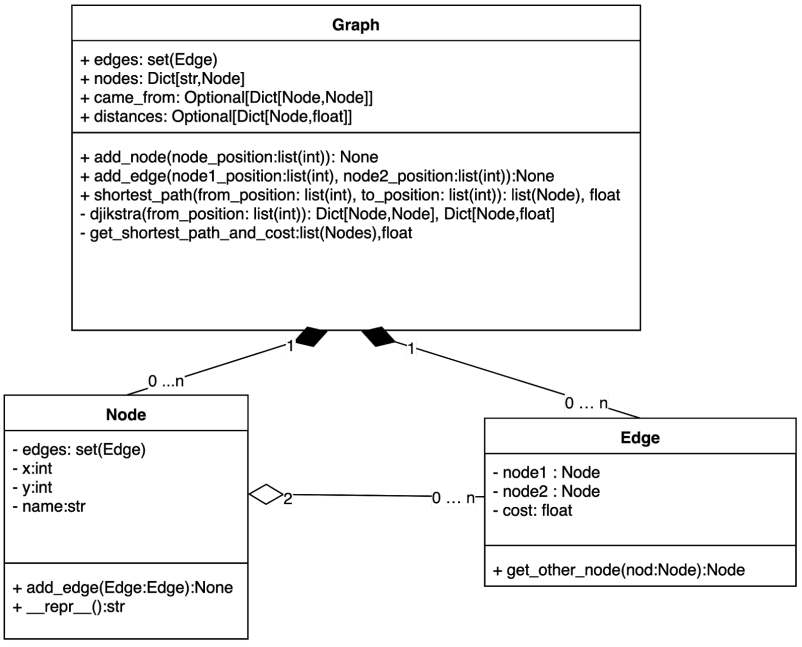
Bei der Entwicklung habe ich viele Sachen ausprobiert und getestet.
Da ich sehr oft aus Versehen mein komplettes Programm zerstört habe und plötzlich nichts mehr funktioniert hat, habe ich die Versionskontrolle **Git** und Github verwendet.
Durch Git hatte ich die Möglichkeit einfach zu alten Ständen zurück zu springen, wenn nichts mehr funktioniert hat. So hatte ich mehr Mut Sachen auszuprobieren, da ich wusste, dass ich immer zu alten Versionen zurück springen kann.

Für die Entwicklung habe ich die **IDE PyCharm** von JetBrains verwendet (die man als Schüler kostenlos bekommt). Für die einfache Verwendung von Git habe ich das Git Plugin von PyCharm verwendet.

Implementation:

UML DIAGRAMM:

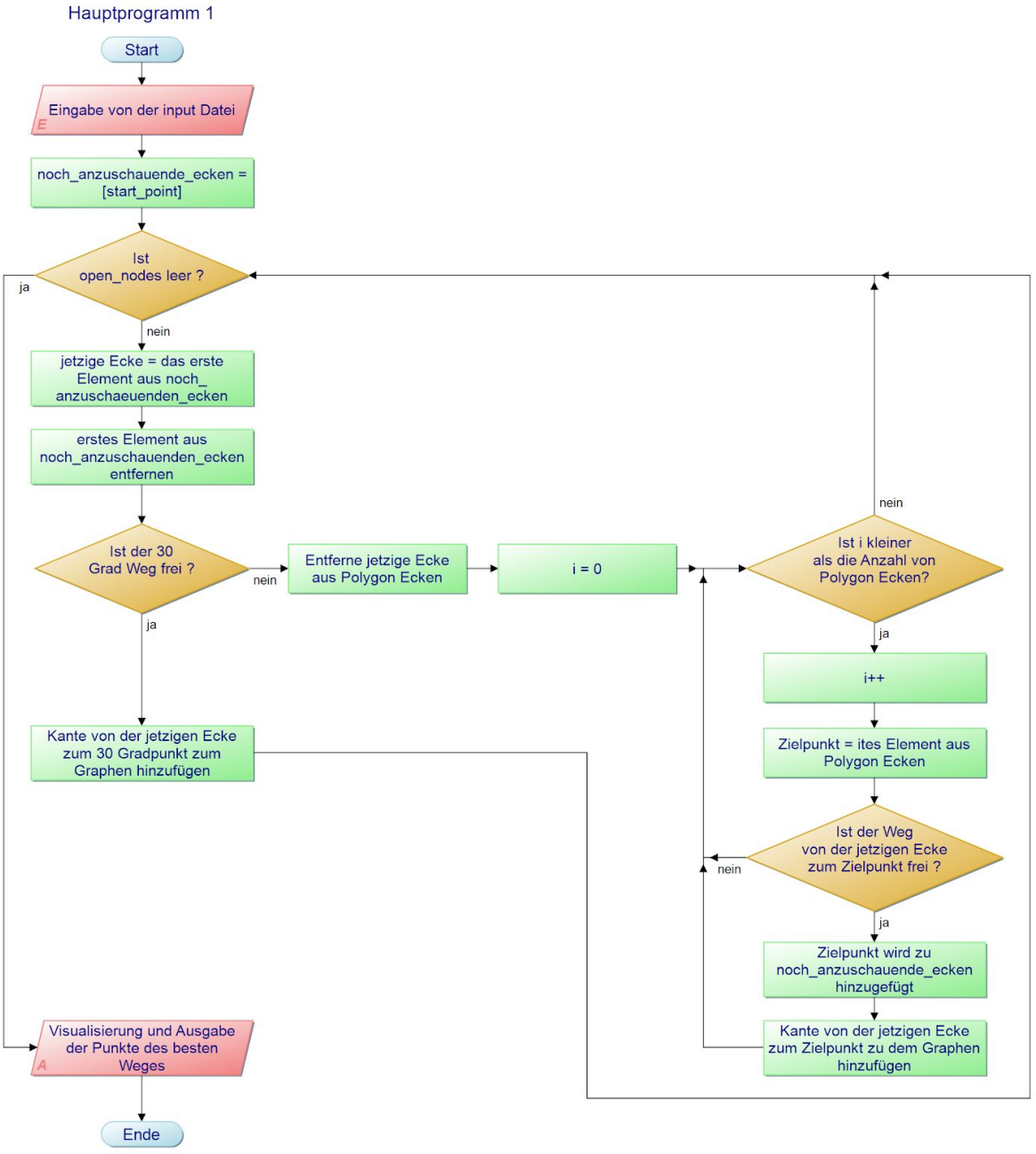
Klassen für den Graphen:



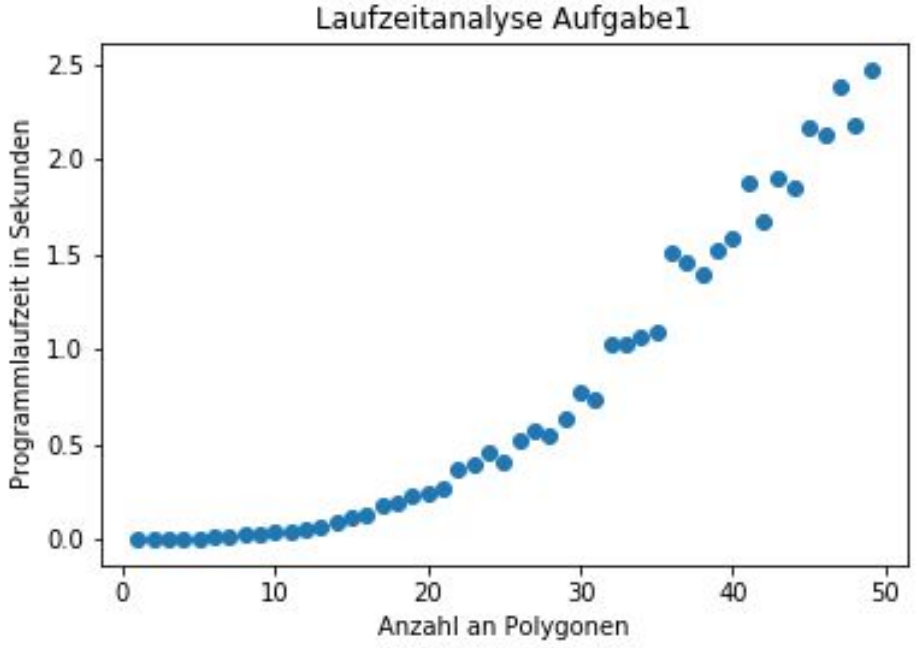
Dateistruktur:

Die Hauptlogik befindet sich in der Datei **aufgabe1.py**. Help-Funktionen befinden sich in **helper_functions_aufgabe1.py**, sowas wie, ob eine Linie sich mit einem Polygon schneidet.
Alle Sachen für den Graphen und djikstra Algorithmus finden sich im Ordner **graph_theory**(Graph,Node,Edge,HeapEntry). Im Ordner **Animation** befinden sich die Klassen Bus, Lisa, Polygon, diese werden hauptsächlich für die Animation gebraucht. Im Ordner Input befinden sich die Input Dateien, von der BwInf Seite.

PAP des Algorithmuses:



Laufzeitanalyse:



**Es wurden Durschnittswerte gebildet von den Programmlaufzeiten,bei zufälligen Polygonen

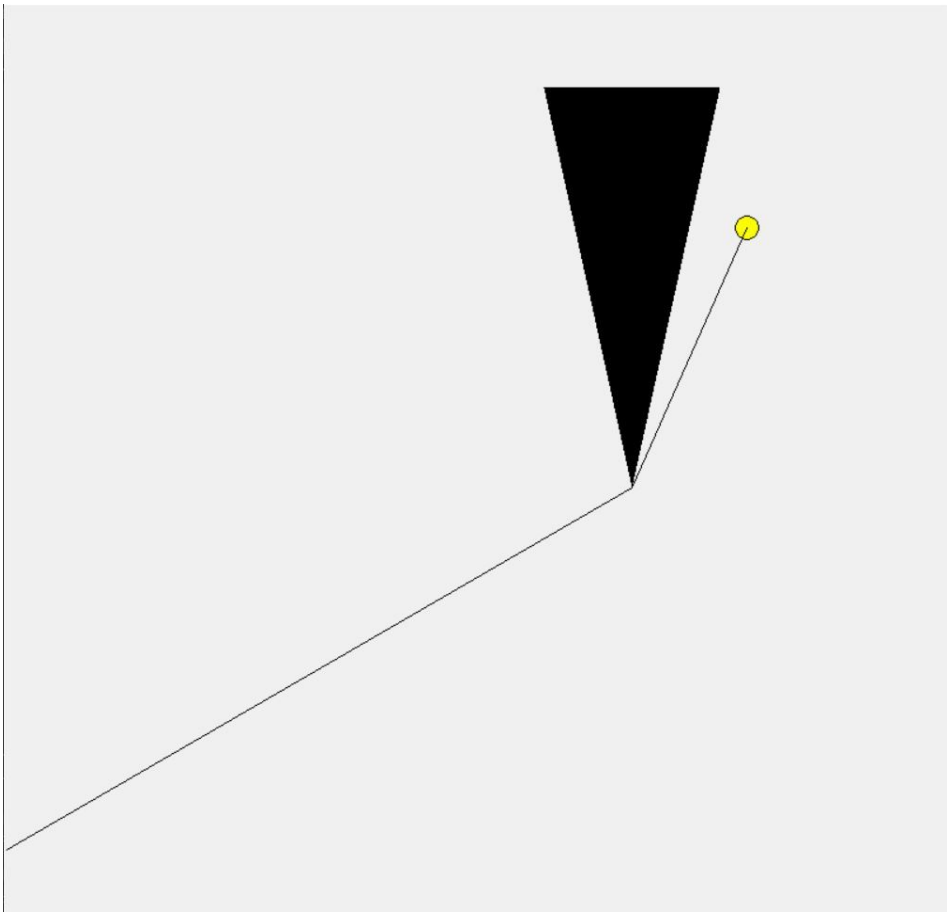
In diesem Graph kann man sehen, dass die Zeit sich polynomisch vergrößert.
Genauer gesagt **O**(**e^2** * **p**) worst case , mit **e** Ecken und **p** Polygonen.

Beispiel Ausgaben:

lisarennt1.txt:

Ausgabe:
Startzeit;Treffen auf den Bus;Zeit für Lisa’s Route; Länge Lisa’s Route in Meter; Hindernisse

('7:27:59', '7:31:26', '0:03:26', 860, [((633, 189), 'L'), ((535, 410), 'P1'), ((0, 719), 'Straße')])
Visualisierung:

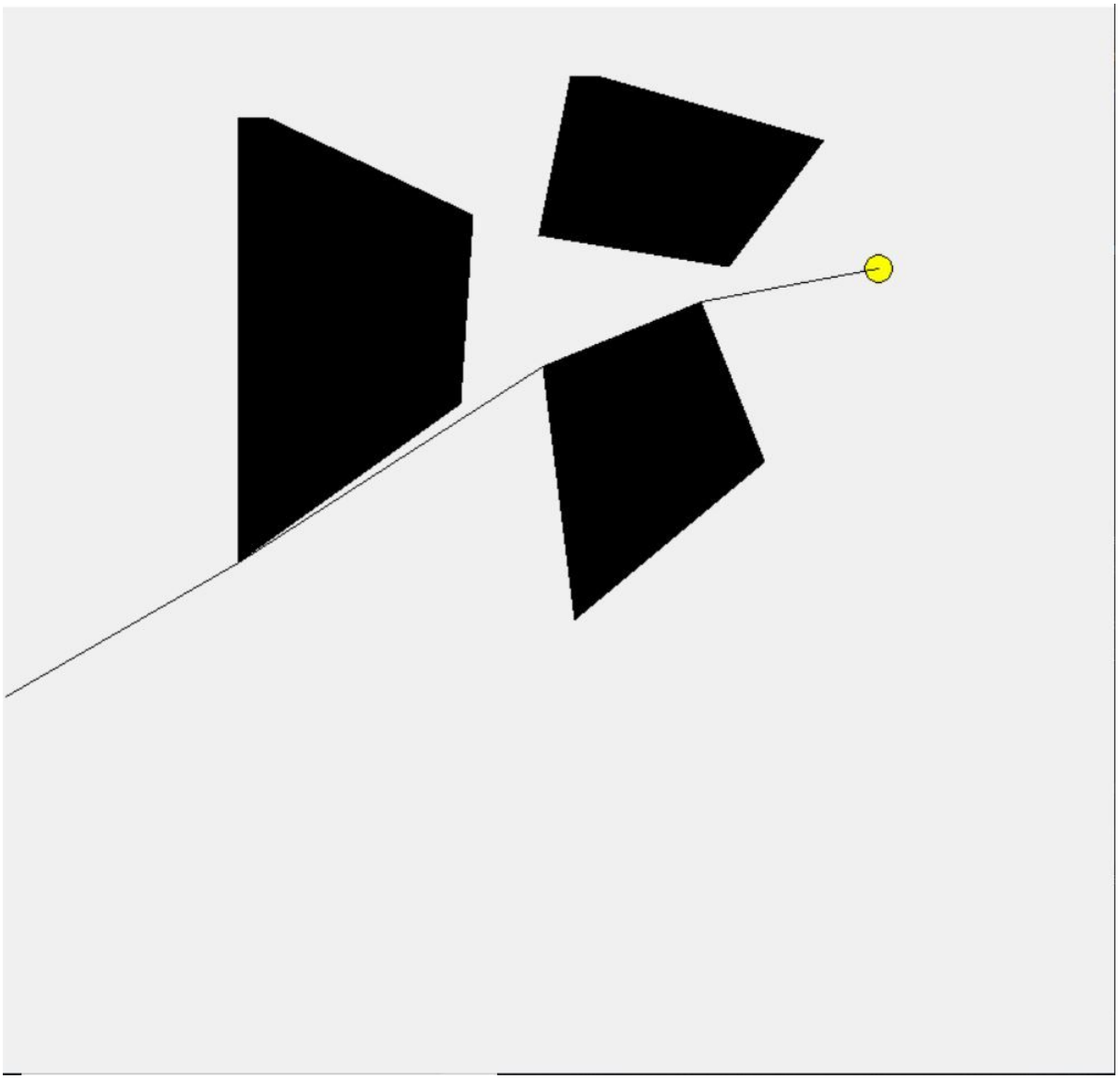


lisarennt2.txt

Ausgabe:

('7:28:8', '7:31:0', '0:02:51', 713, [((633, 189), 'L'), ((505, 213), 'P1'), ((390, 260), 'P1'), ((170, 402), 'P3'), ((0, 500), 'Straße')])

Visualisierung:

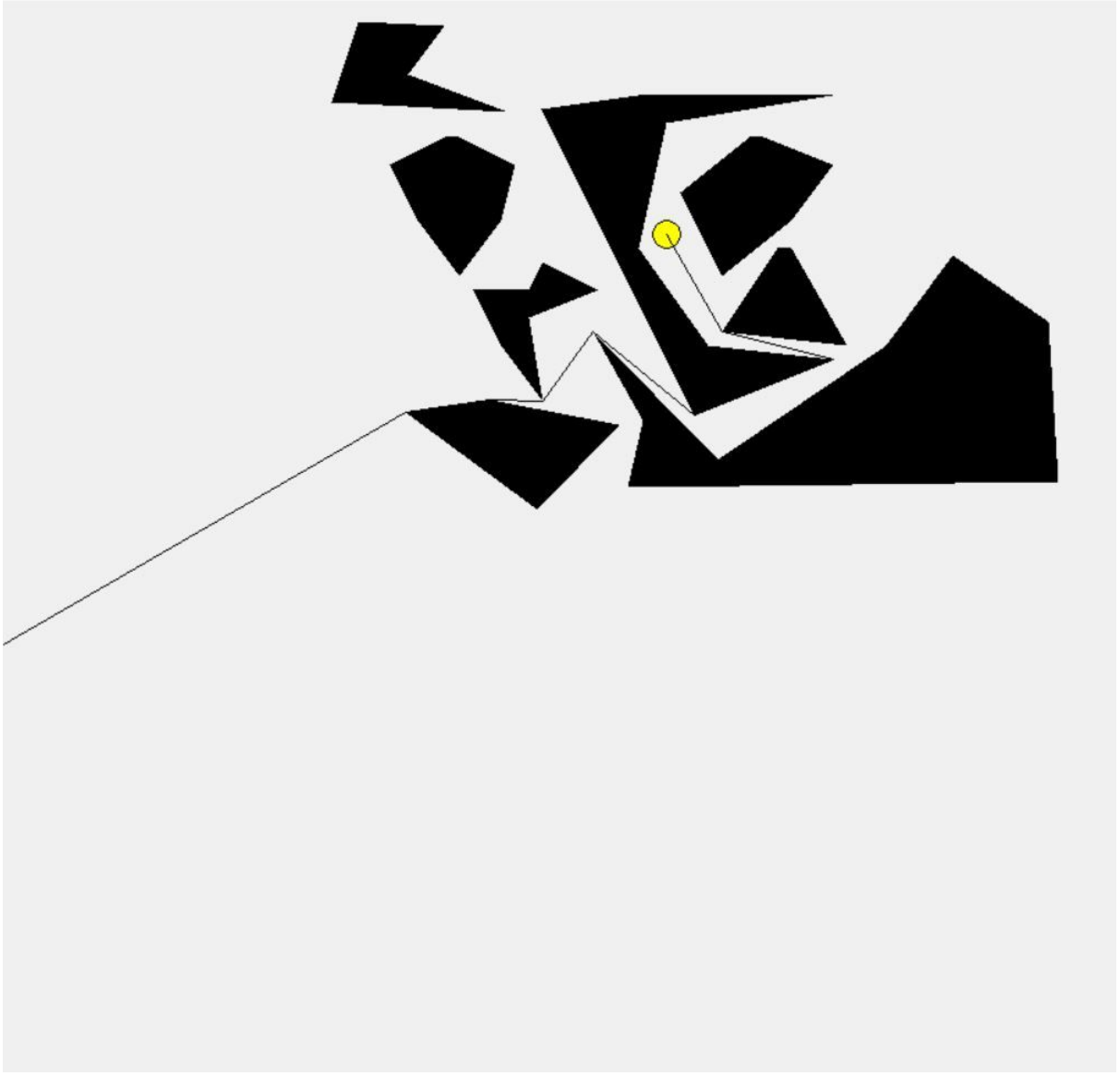


lisarennt3.txt

Ausgabe:

('7:27:28', '7:30:55', '0:03:27', 863, [((479, 168), 'L'), ((519, 238), 'P2'), ((599, 258), 'P3'), ((499, 298), 'P3'), ((426, 238), 'P8'), ((390, 288), 'P5'), ((352, 287), 'P6'), ((291, 296), 'P6'), ((0, 464), 'Straße')])

Visualisierung:

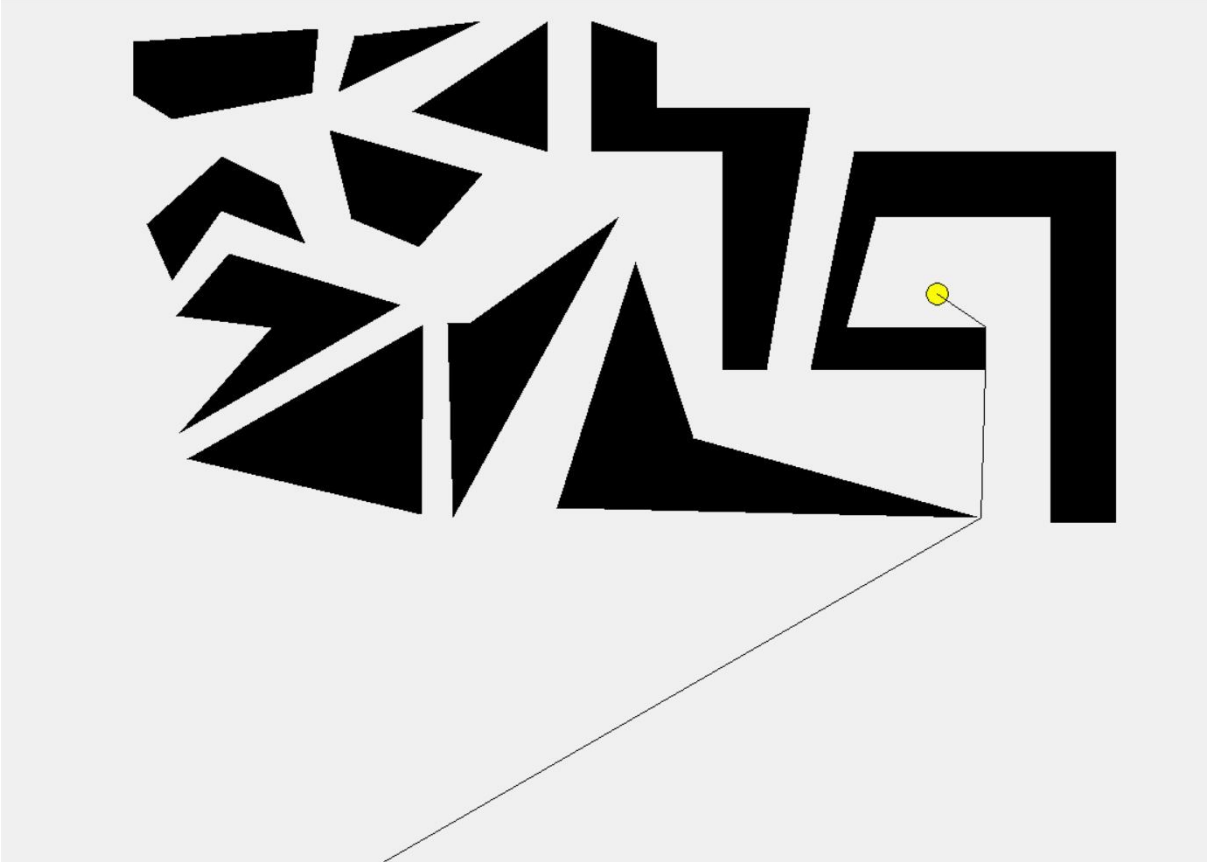


lisarennt4.txt

Ausgabe:

('7:26:55', '7:31:59', '0:05:03', 1263, [((856, 270), 'L'), ((900, 300), 'P11'), ((900, 340), 'P11'), ((896, 475), 'P10'), ((0, 992), 'Straße')])

Visualisierung:

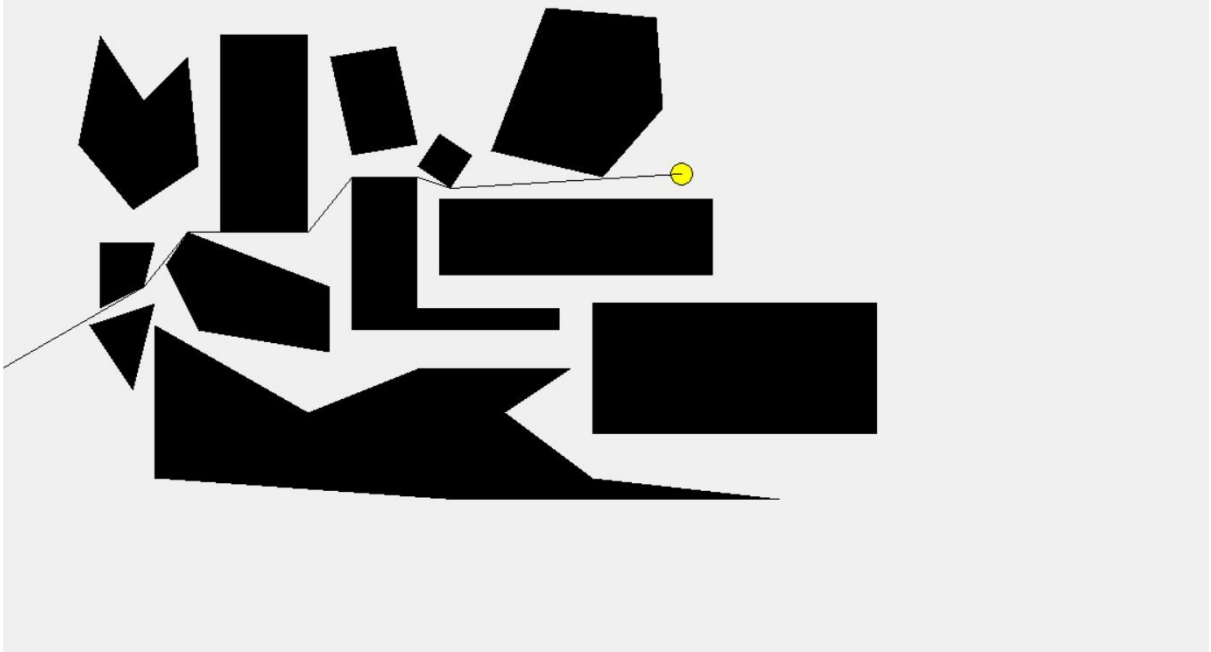


lisarennt5.txt

Ausgabe:

('7:27:54', '7:30:40', '0:02:46', 691, [((621, 162), 'L'), ((410, 175), 'P8'), ((380, 165), 'P3'), ((320, 165), 'P3'), ((280, 215), 'P5'), ((170, 215), 'P6'), ((130, 265), 'P9'), ((0, 340), 'Straße')])

Visualisierung:

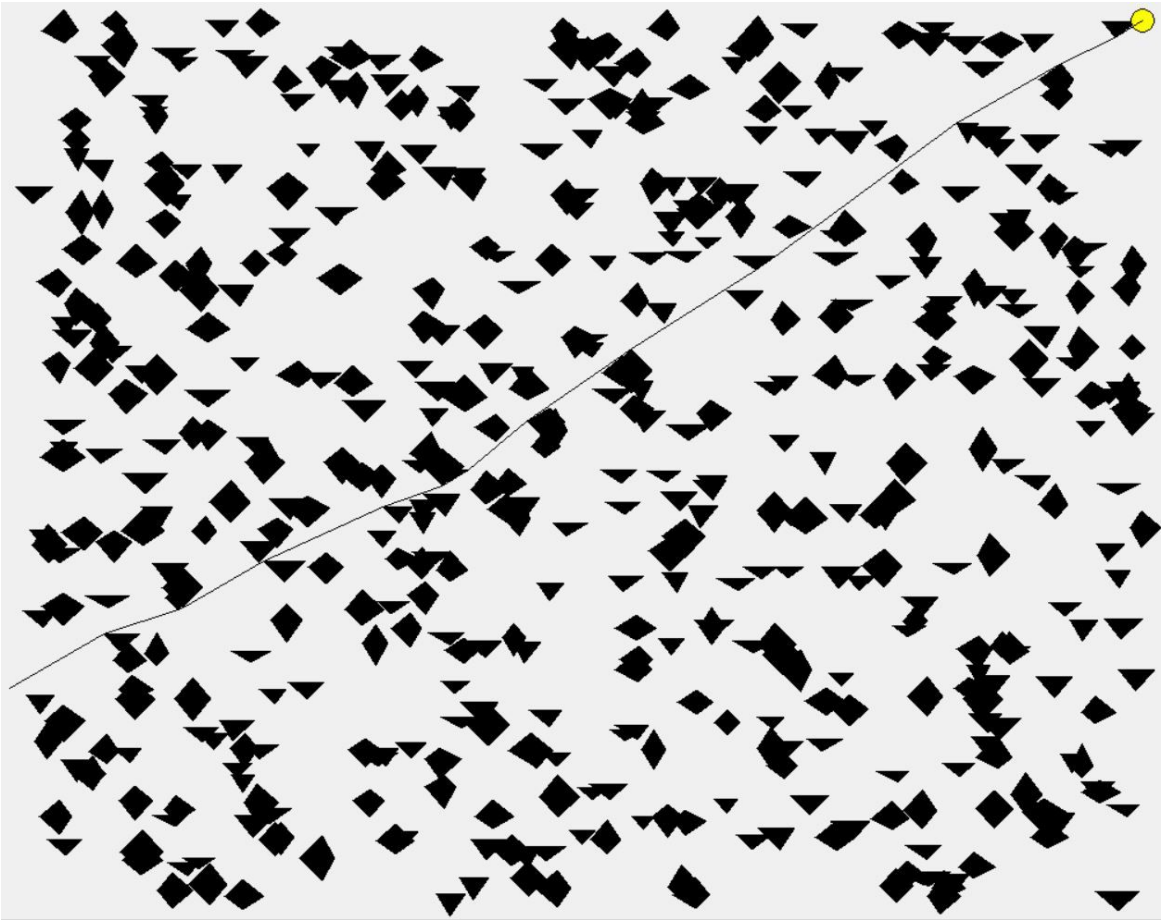


Zufällig 500 generierte Polygone :

Ausgabe:

('7:26:37', '7:31:12', '0:04:35', 1144, [((980, 22), 'L'), ((956, 37), 'P66'), ((910, 59), 'P460'), ((819, 111), 'P94'), ((663, 226), 'P286'), ((646, 238), 'P286'), ((539, 306), 'P45'), ((448, 369), 'P328'), ((398, 411), 'P360'), ((375, 425), 'P65'), ((324, 443), 'P205'), ((221, 490), 'P26'), ((152, 530), 'P218'), ((147, 532), 'P145'), ((83, 553), 'P474'), ((0, 601), 'Straße')])

Visualisierung:



Informationen zu meinem Programm:

Ich habe das Programm mit **Python** (Version 3.7) programmiert. Zusätzlich habe ich die Bibliothek **shapely** verwendet, um zu testen, ob sich eine Linie mit einem Polygon schneidet. Es kann sein, dass sie shapely installieren müssen um das Programm zu starten. Wenn sie **pip** haben können sie einfach **pip install shapely** eingeben.
Das Haupt-Programm befindet sich in der main.py Datei. Wenn die **main.py** Datei gestartet wurde werden sie aufgefordert die Polygon Datei einzugeben. Zum Beispiel “lisarennt1.txt”.

Quellen:

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://en.wikipedia.org/wiki/Heap_%28data_structure%29
https://en.wikipedia.org/wiki/Priority_queue
https://en.wikipedia.org/wiki/Big_O_notation
https://en.wikipedia.org/wiki/Python_%28programming_language%29
https://stackoverflow.com/
https://en.wikipedia.org/wiki/Graph_(abstract_data_type)