

Kurs-ID:	BA-INF 051
Kurs:	Projektgruppe Maschinelles Lernen
Semester:	Winter 2021/22
Betreuer/-in:	Florian Seiffarth
Studierende:	Robert Schmidt: 3358717 Luca Eichler: 3357021 Linus Römke 3202240



Projektgruppe Maschinelles Lernen

Thema 4: Generierung einer Datenbank von Straßennetzwerken

Robert Schmidt Luca Eichler Linus Römke

20. März 2022

Zusammenfassung

Die Aufgabe unserer Projektgruppe war es, ein Programm zu implementieren, welches je nach angegebenen Eigenschaften, eine Graphendatenbank von verschiedenen Straßennetzwerken erstellen soll. Dabei sollen Graphen und Knoten gelabelt werden. Solche Daten könnten beispielsweise als Trainingsdaten dienen, um einen Klassifikator zu trainieren. Dieser könnte beispielsweise versuchen, Städte anhand von Graphen zu erkennen. Um dieses Programm zu implementieren, haben wir OS-Mnx und, um es benutzerfreundlich mit einem GUI umzusetzen, PySide2 verwendet. Das Ergebnis ist ein flexibles Programm das über ein GUI bedienbar und je nach Bedürfnissen adaptierbar ist. Es ermöglicht das Herunterladen von Städten als Graphen. Man kann aus diesen Teilgraphen erstellen und nach Eingabe Labeln.

Linus

1 Einleitung

Mittels maschinellem Lernens werden bereits verschiedenste alltägliche Probleme gelöst. Dies gilt auch in Bezug auf geographische Daten. Um maschinelles Lernen anwenden zu können, werden umfangreiche, bereinigte und einheitliche Daten benötigt. Unser Projekt setzt da an und befasst sich mit dem Generieren von Daten, in Form von Graphen, welche Straßennetzwerke darstellen. Unser Projekt überlässt dem Nutzer umfangreiche Eingabemöglichkeiten um ein möglichst breites Spektrum an generierbaren Daten zu bieten und so auch für unvorhergesehene Anwendungsfälle geeignet zu sein. Dabei haben wir uns auf die Python Bibliothek OSMnx gestützt, wobei es der erste Teil der Projektgruppe war, uns in diese einzuarbeiten.

Linus

Das Programm soll, je nach angegebenen Eigenschaften, Graphen von verschiedenen Straßen-Netzwerken herunterladen, aus diesen Teilgraphen generieren und diese dann labeln können. Die Dateien sollen strukturiert gespeichert werden. Das Generieren gibt einem die Möglichkeit, aus einer Stadt mehrere Graphen mit gleichen Eigenschaften, wie Beispielsweise die Anzahl der Knoten oder Kanten, zu erstellen. Dem Nutzer wird überlassen, welche Einrichtungen beim Labeln des Graphen beachtet werden. Das Programm soll zu jeden Graphen Statistiken berechnen und auch über alle Graphen einen Durchschnitt der Statistiken berechnen. Jeder Schritt, also das Downloaden, Generieren und Labeln, lässt sich auch einzeln oder in jeder Möglichen Kombination durchführen. Um die Benutzung zu vereinfachen wird ein User Interface zur Verfügung gestellt.

2 OSMnx

OSMnx ist eine Software-Bibliothek, welche in Python geschrieben wurde. Entwickelt von Geoff Boeing, erlaubt es OSMnx Straßennetzwerke von OpenStreetMap herunterzuladen, zu modellieren, analysieren und visualisieren.

Der Name setzt sich zusammen aus OpenStreetMap und NetworkX, denn OSMnx basiert auf NetworkX, Geopandas und Matplotlib. Des weiteren verwendet es die API's von OpenStreetMap um die nötigen Daten herunterzuladen. In OSMnx werden die rohen OpenStreetMap-Daten, also die heruntergeladenen Straßennetzwerke, als Graphen modelliert. Dabei ist ein Knoten immer eine Kreuzung von zwei Straßen oder das Enden einer Straße als Sackgasse. Kanten hingegen repräsentieren immer die Straßen.

2.1 Grundlegende Funktionen von OSMnx

OSMnx lässt sich vielseitig für beispielsweise folgende Funktionen verwenden:

- Straßennetzwerke der ganzen Welt herunterladen und modellieren
- Straßennetzwerke simplifizieren und bereinigen
- Graphen speichern und einlesen
- Graphen analysieren und visualisieren
- Graphalgorithmen verwenden
- Verschiedene Infrastruktur und Points of Interest herunterladen

Für uns eine grundlegend sehr wichtige Funktion von OSMnx ist das Herunterladen der Straßennetzwerke. Dabei kann man das bereits mit nur einer Zeile Code umsetzen:

```
Graph = osmnx.graph_from_place(query, parameter)
```

Dabei hat man die Wahl, ob man Orte oder eine Liste von Orten, Koordinaten mit einem Radius, eine Adresse mit einem Radius oder eine Bounding Box bzw. ein Polygon angibt, um das Straßennetzwerk herunterzuladen. Als Nutzer kann man angeben, welche Art von Straßen, wie zum Beispiel Fußgängerwege oder Einbahnstraßen, man herunterladen möchte.

OSMnx bereinigt und simplifiziert die Graphen nach dem Download automatisch, wobei man auch die Möglichkeit hat das zu unterlassen. Dabei werden Nodes die direkt nebeneinander sind, zu einer Node vereint um Nodecluster zu vermeiden. Des weiteren werden Nodes, die keine Kreuzung darstellen, entfernt, ohne jedoch die geometrischen Daten zu verlieren. Dabei werden die Kanten entsprechend zusammengefügt. Solche Knoten, die keine Kreuzung darstellen, sind in den OpenStreetMap Daten vorhanden, da Knicke in Straßen auch als Knoten umgesetzt werden.

Das OSMnx.plots Modul erlaubt einem das Visualisieren von Graphen. Dabei hat man umfangreiche Möglichkeiten die Farben sowie Darstellung der Graphen anzupassen. Diese stützt sich auf die Matplotlib Bibliothek. Ein Beispiel für das Plotten in OSMnx sowie einen der Graphenalgorithmen sieht man in Abbildung 1.

```
#Berechnet eine Route vom Institutgebäude zum Brückenforum
route = ox.shortest_path(G, nodes[0], nodes[1], weight="length")

#Plote die zusätzlich die Route
fig, ax = ox.plot_graph_route(G, route, route_color='blue', route_alpha=1, route_linewidth=1.5, node_size=30, ...)
```



Abbildung 1: Route in Graph geplottet

In diesem Beispiel wird die NetworkX Funktion `shortest_path` genutzt um eine Route von einem Punkt zu einem anderen zu bestimmen, und anschließend wird die `plot_graph_route` Funktion verwendet, um diesen Weg zu visualisieren.

2.2 NetworkX

OSMnx baut auf dem Pythonpaket NetworkX auf. Sie wird in OSMnx, unter anderem für die Graphen-Datenstrukturen, verwendet. Die Klasse "MultiDiGraph", welche von OSMnx verwendet wird, erlaubt es, gerichtete Graphen mit Mehrfachkanten und Schleifen umzusetzen. Dabei wird eine Einbahnstraße immer mit einer Kante zwischen zwei Knoten repräsentiert. Wenn jedoch beide Richtungen erlaubt sind, werden zwei Kanten zwischen den Knoten verwendet, um jeweils beide Richtungen zu ermöglichen. Wenn man ungerichtete Graphen präferiert, erlaubt es OSMnx auch MultiDiGraphs in MultiGraphs umzuwandeln. Des weiteren nutzt OSMnx die verschiedenen Graphenalgorithmen, die in NetworkX umgesetzt sind, um zum Beispiel die kürzesten Wege zu bestimmen.

3 Generieren der Graphen

Einer der Schwerpunkte unseres Projektes ist das Generieren neuer Graphen als Trainingsdaten. Anschaulich gesprochen wird dabei ein Teil des ursprünglichen Graphen „herausgeschnitten“. Ein Beispiel für einen aus der Stadt Köln generierten Graphen ist in Abbildung 2 zu sehen.

3.1 Breitensuche

Um aus dem Ursprungsgraphen einen neuen Teilgraphen zu erzeugen, bietet es sich an, eine explorative Suche auf dem Graphen durchzuführen. Alle gefundenen Knoten und Kanten sollen dem Graphen hinzugefügt werden. Dadurch kann sichergestellt werden, dass nur Knoten bzw. Kanten, die zuvor schon im Graphen enthalten waren, zum neuen Graphen hinzugefügt werden. Dies hat außerdem den Nebeneffekt, dass unsere Graphen zusammenhängend sind. Naheliegende Möglichkeiten, solch eine Suche zu implementieren, sind Tiefen- oder Breitensuche. Wir haben uns entschieden, eine Breitensuche zu verwenden. Das Verhalten der Tiefensuche haben wir nicht untersucht, es wäre sicherlich interessant, sie mit der Breitensuche zu vergleichen. Zu beachten ist noch, dass unsere Breitensuche auch Knoten expandiert, die nur über eingehende Kanten mit dem aktuellen Knoten verbunden sind (normal: nur ausgehende Kanten). Der Grund für diese Änderung ist, dass Knoten mit Outgrad 0 (also ohne ausgehende Kanten) in den heruntergeladenen Graphen vorkommen können. Diese Knoten repräsentieren meistens den Endpunkt einer Sackgasse.



Abbildung 2: Generierter Graph aus Köln, 1000 Knoten

Unser Programm ermöglicht es, die Knoten- oder Kantenzahl der generierten Graphen zu wählen. Es werden also solange weitere Kanten und Knoten gesucht, bis entweder das

Knoten- oder Kantenlimit erreicht ist.

3.2 Prozentuale Breitensuche

Werden Knoten- oder Kantenzahlen als Endkriterium für die Breitensuche genutzt, so kann es passieren, dass der ursprüngliche Graph weniger Knoten bzw. Kanten hat als die gewählte Kanten- oder Knotenzahl. In diesem Fall wird terminiert, sobald der gesamte Graph heruntergeladen wurde. Allerdings könnte es je nach Anwendungszweck für unsere Trainingsdaten nicht erwünscht sein, dass der gesamte Graph oder ein sehr großer Anteil davon generiert wird, da sich die generierten Graphen dann kaum voneinander unterscheiden. Deshalb haben wir uns für ein weiteres optionales Kriterium entschieden, nach der die Breitensuche terminiert. Hiermit ist die Möglichkeit gemeint, einen Anteil in Prozent des Ursprungsgraphen anzugeben, der generiert werden soll. Es kann auch bei dieser Variante wieder nach Knoten- bzw. Kantenzahl entschieden werden, wann die Breitensuche terminiert.

3.3 Ausläufer bei der Breitensuche

Beim Testen der von uns implementierten Breitensuche auf verschiedenen Städtegraphen ist uns aufgefallen, dass die generierten Graphen häufig weite Ausläufer enthielten. Damit sind Teile des Graphen gemeint, welche verglichen mit dem Rest des Graphen eine viel weitere Entfernung vom Startpunkt der Breitensuche haben.

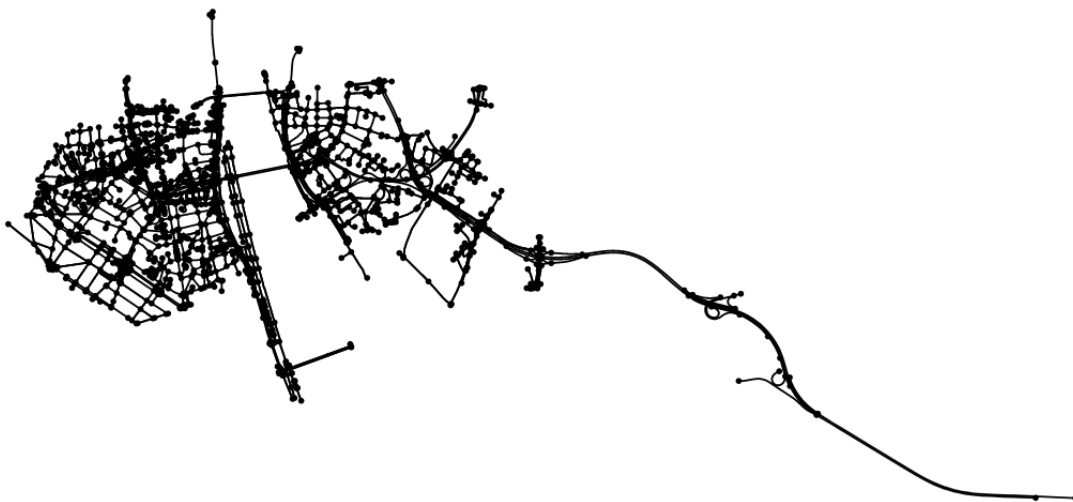


Abbildung 3: Ausläufer bei der normalen Breitensuche

Die Ausläufer sind häufig einzelne Straßen, deren angrenzenden Straßen gar nicht oder nur unvollständig von der Breitensuche besucht wurden. Dieses Verhalten entsprach nicht unserer Erwartung. Da wir eine Breitensuche genutzt haben, sind wir stattdessen davon ausgegangen, dass die Graphen sehr viel gleichmäßiger um den Startknoten herum generiert werden würden. Nach genauerer Untersuchung der Graphen mit weiten Ausläufern

haben wir herausgefunden, dass die Ausläufer durch überdurchschnittlich lange Kanten erzeugt werden. Meistens wurden diese nämlich durch Autobahnen erzeugt, wo eine Kante meist die gesamte Strecke zwischen zwei Ausfahrten repräsentiert. Im Vergleich sind die Strecken innerhalb der Stadt wesentlich kürzer (meistens wenige Meter). Die starke Diskrepanz der Kantenlänge führt also schon nach dem Expandieren weniger Knoten auf einer Autobahn zu großen Ausläufern.

3.4 Kompakte Breitensuche

Hinweis: Wenn in diesem und in den folgenden Abschnitten von Kompaktheit bzw. kompakten Graphen gesprochen wird, so ist nicht die verbreitete Definition der Kompaktheit eines Graphen gemeint. Viel mehr hat die Kompaktheit hier eine geometrische Bedeutung. Ein Graph, der möglichst gleichmäßig um den Startknoten erzeugt wurde und somit keine oder kaum leere „Zwischenräume“ oder „Lücken“ enthält, wird im Folgenden kompakt genannt.

Für die Nutzung als Trainingsdaten könnte das im vorigem Abschnitt beschriebene Verhalten unerwünscht sein. Aus diesem Grund haben wir eine modifizierte Breitensuche entworfen, welche kompaktere Graphen (beachte Hinweis) erzeugen soll. Die neue Variante der Breitensuche muss nun also auch geometrische Parameter der Knoten (anstatt nur topologische) berücksichtigen. Eine möglicher Ansatz wäre, priorisiert Knoten zu expandieren, deren Verbindungskante zum aktuellen Knoten die Kürzeste ist. Allerdings würden dann nur noch die kürzesten Kanten expandiert werden und lange Kanten, welche geometrisch gesehen näher am Startknoten liegen, würden übersprungen werden. Deshalb haben wir diese Idee verworfen. Stattdessen haben wir uns für eine Priorisierung anhand der euklidischen Distanz zum Startknoten entschieden. Es wird also immer derjenige Knoten zum neuen Graphen hinzugefügt, der mindestens eine Verbindungskante zu einem bereits im Graphen existierenden Knoten hat und dessen euklidische Distanz zum Startknoten die Kürzeste unter allen möglichen Knoten ist. Die Implementation wurde mittels der Python-Datenstruktur „Priority Queue“ realisiert. Diese erlaubt es, jedem Element eine Priorität zuzuweisen. Beim Entnehmen eines Elementes wird stets das Element mit dem kleinsten Prioritätswert zurückgegeben. Also bietet es sich an, als Priorität für einen Knoten direkt die euklidische Distanz zum Startknoten zu übergeben. Durch diese Änderung werden die Graphen bereits viel kompakter.



Abbildung 4: Mittels kompakter Breitensuche generierter Graph aus Köln, 1000 Knoten

3.5 Breitensuche und Kompakte Breitensuche im Vergleich

Mittels der modifizierten Breitensuche können kompaktere Graphen aus den Ursprungsgraphen erzeugt werden. Im Folgenden wird ein von uns durchgeführtes Experiment beschrieben in dem untersucht wurde, wie viel kompakter die mit der neuen Methode erzeugten Graphen sind. Als Ursprungsgraph haben wir den gesamten Graphen der Stadt Köln verwendet, dessen Knotenanzahl in etwa 100.000 Knoten beträgt. Es wurden 3 verschiedene Tests durchgeführt. In jedem Test wurden jeweils 100 Graphen mit den beiden Methoden generiert. Dabei unterschieden sich die Tests in der Größe der generierten Graphen. Die jeweiligen Größen nach Knotenzahl betrugen 100, 1000 und 10000 Knoten. Der Vergleich der Kompaktheit ist mittels der vom Graphen eingeschlossenen Fläche durchgeführt worden. Die eingeschlossene Fläche wurde berechnet, indem wir die minimale Axis Aligned Bounding Box des betrachteten Graphen berechnet haben. Eine andere Möglichkeit ist, die konvexe Hülle der erzeugten Graphen zum Vergleichen der Kompaktheit zu nutzen, um noch genauere Ergebnisse zu erhalten. Einen solchen Test haben wir allerdings nicht für notwendig gehalten, da der durchgeführte Test bereits relativ eindeutige Ergebnisse liefert.



Abbildung 5: Kompakte und normale Breitensuche im Vergleich. Türkis: Kompakte Breitensuche, Hellblau: Normale Breitensuche

Wir haben für die drei verschiedenen Tests eine Verbesserung von jeweils 38% (100 Knoten), 66% (1000 Knoten) und 81% (10000 Knoten) berechnet. In allen drei Tests kann man also eine deutliche Verringerung der vom Graphen eingeschlossenen Fläche beobachten. Des Weiteren ist auffällig, dass die verglichenen Flächen immer mehr von einander abweichen, je größer die generierten Graphen sind. Dies begründen wir einerseits mit der erhöhten Wahrscheinlichkeit, bei der Breitensuche auf einen Autobahnknoten zu treffen, wenn mehr Knoten expandiert werden. Andererseits wird für jeden weiteren auf einer Autobahn expandierten Knoten die Fläche des erzeugten Graphen um ein Vielfaches größer. Je größer die Knotenzahl, desto mehr „Autobahnkanten“ können expandiert werden. Allerdings sollte noch beachtet werden, dass dieser Test stark von dem benutzten Graphen abhängig ist. Bei der Durchführung des Tests auf anderen Städten würde man möglicherweise eine viel kleinere Verringerung der eingeschlossenen Fläche beobachten.

3.6 Radius und Bounding Box

Neben der kompakten Breitensuche haben wir noch zwei weitere mögliche Modi implementiert, mit welchen Graphen erzeugt werden können. Beide Modi basieren ebenfalls auf der Breitensuche. Ihre Funktionsweise wird im Folgenden beschrieben und lässt sich bereits am Namen ableiten.

Radius: Die Breitensuche expandiert all erreichbaren Knoten innerhalb eines gegebenen Radius um den Startknoten herum.

Bounding Box: Die Breitensuche expandiert alle erreichbaren Knoten innerhalb einer Axis Aligned Bounding Box gegebener Höhe und Breite, dessen Mittelpunkt der Startknoten ist.

Diese beiden Modi könnten interessant sein, falls man sich beim Trainieren eines Klassifikators eher für geometrische Eigenschaften eines Graphen wie die Dichte der Straßen innerhalb des Netzwerkes interessiert. Beispielsweise könnte man zwei Graphen aus einer urbanen und einer ländlichen Gegend mit derselben Bounding Box generieren. Die urbane Gegend hat hier erwartungsgemäß eine viel höhere Straßendichte (Anzahl Straßen in Relation zu der eingeschlossenen Fläche). Anhand der vorangegangenen Beschreibung wird außerdem deutlich, dass die beiden Modi sich sehr ähneln. Der einzige Unterschied ist, dass entweder kreisförmige oder rechteckige Graphen erzeugt werden.

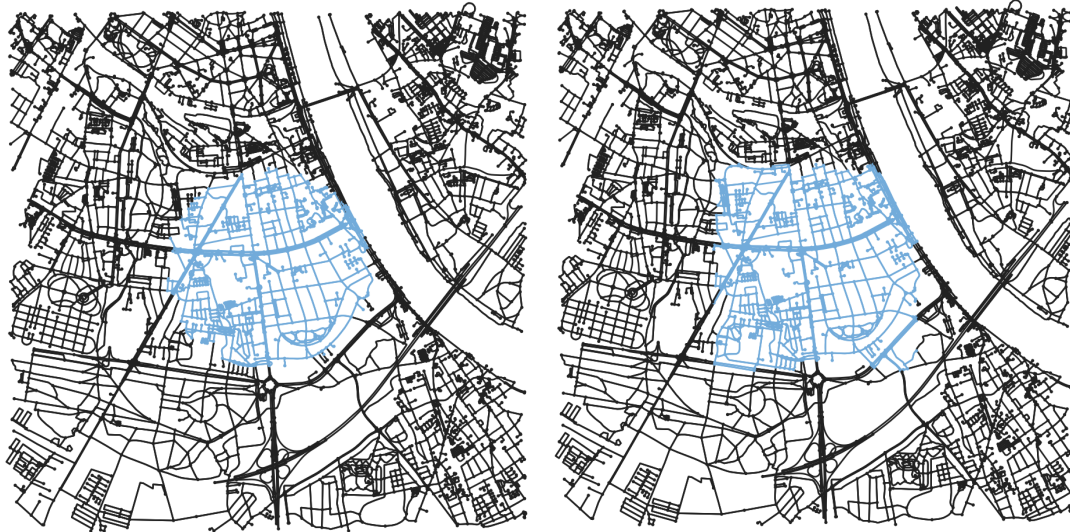


Abbildung 6: Mittels Radius und Bounding Box generierte Graphen aus Köln, 100 Knoten. Links: Radius (1000m), Rechts: Bounding Box (2000m, Breite und Höhe)

In der obigen Abbildung können die kreisförmigen bzw. rechteckigen Formen der generierten Graphen gut erkannt werden. Unsere Implementierung bietet auch für diese Modi die Möglichkeit, eine maximale Kanten- oder Knotenzahl der erzeugten Graphen anzugeben. Im fertigen Programm werden jedoch standardmäßig die Gesamtknotenanzahl bzw. die Gesamtkantenanzahl des Ursprungsgraphen übergeben. Diese Parameter haben also keinen Einfluss mehr, es wird standardmäßig der gesamte Kreis oder die ganze Bounding Box ausgefüllt.

3.7 Zeitkomplexität der verschiedenen Methoden

In diesem Abschnitt wollen wir die Zeitkomplexitäten der vier vorgestellten Modi miteinander vergleichen. Es bezeichne n die Knotenanzahl und m die Kantenanzahl eines erzeugten Graphen. Die Laufzeit einer Breitensuche beträgt $\mathcal{O}(n + m)$. Dies ist auch bei unserer Implementation der Breitensuche der Fall, die Laufzeit wächst also linear mit der Größe des generierten Graphen. In Straßennetzwerken stehen die Kanten- und Knotenanzahlen meist in einem konstanten Verhältnis zueinander. Das liegt daran, dass Knoten Kreuzungen (oder Straßenenden) repräsentieren und somit meist nicht mehr als vier ausgehende Kanten haben. Tatsächlich hat sich bei einem Test ergeben, dass fast alle Städte ein Verhältnis von Knotenanzahl zu Kantenanzahl zwischen 1:2,5 und 1:3 haben. In unserem Fall ist m also immer in derselben Größenordnung wie n und statt $\mathcal{O}(n + m)$ können wir deshalb $\mathcal{O}(n)$ schreiben. Für die Modi Radius und Bounding Box gilt zwar theo-

retisch dieselbe Zeitkomplexität wie für die Breitensuche, allerdings kommt noch eine weitere Variable dazu, nämlich der Radius bzw. die Größe der Bounding Box. Dies erschwert den Vergleich mit den anderen Methoden. In den meisten Fällen sind die beiden Methoden etwas schneller als die kompakte Breitensuche und um einiges langsamer als die normale Breitensuche. Dies liegt an den zusätzlichen geometrischen Überprüfungen, welche testen ob der betrachtete Knoten noch innerhalb der Bounding Box oder innerhalb des Radius ist. Der Unterschied der Laufzeiten von der Breitensuche mittels Radius und der Breitensuche mittels Bounding Box ist aufgrund der ähnlichen Implementierung beider Methoden vernachlässigbar und es ist stark abhängig vom Graphen, welche Methode die geringere Laufzeit hat. Die kompakte Breitensuche nutzt eine Priority Queue, welche zum Einfügen Zeit in der Größenordnung $\mathcal{O}(\log n)$ benötigt. Da im Worst-Case n Elemente eingefügt werden, ergibt sich als Zeitkomplexität $\mathcal{O}(n \log n)$. Dies macht sich auch in der Praxis bemerkbar. Für einen Graphen mit ungefähr 100.000 Knoten werden in etwa 5 bis 10 Prozent mehr Zeit benötigt, wenn statt der normalen Breitensuche die kompakte Breitensuche verwendet wird. Verglichen mit den anderen Aufgaben unseres Programmes hat das Erzeugen der neuen Graphen allerdings keinen großen Anteil an der Gesamtlaufzeit, weshalb die unterschiedlichen Laufzeiten der verschiedenen Modi sich kaum bemerkbar machen.

4 Graphenlabel

Ein wesentlicher Teil unseres Projektes, ist es die Möglichkeit zu haben, die generierten oder ungenerierten Graphen zu Labeln. Dabei kann der Nutzer eine oder mehrere von sogenannten Amenities spezifizieren. Amenities meint hier verschiedene, für den Nutzer interessante, Einrichtungen in einer Stadt. Beispielsweise könnte man Restaurants, Parkbänke, Bus- und Bahnstationen oder sogar so etwas wie Altglascontainer angeben. Das Programm soll dann den Graphen so anpassen, dass jeder Knoten gelabelt wird und Informationen über die Amenities so gespeichert werden. Linus

4.1 Labelvektoren

Grundlegend hat jeder Knoten des Graphen einen Labelvektor, bei dem dann jeder Eintrag eine vom Nutzer gewählte Amenity repräsentiert. Dabei wird zwischen den Modi „Binary“ und „Counter“ unterschieden. Binary bedeutet, dass jeder Knoten so gelabelt wird, dass nur klar ist **ob** dort eine Amenity ist und nicht wie viele. Der Modus "Counter" speichert für jede Amenity **wie oft** sie bei dem Knoten vorkommt.

4.2 Amenities herunterladen

Mit OSMnx kann man Anfragen an die Overpass API stellen, um points of interest sowie geographische Daten und generelle Informationen von OpenStreetMap herunterzuladen. Dafür erstellen wir zunächst eine Bounding Box um den Graphen, dessen Amenities wir herunterladen wollen. Wie folgt sieht das bei OpenStreetMap aus wenn man innerhalb einer Bounding Box die Amenity "Bench" fragt:

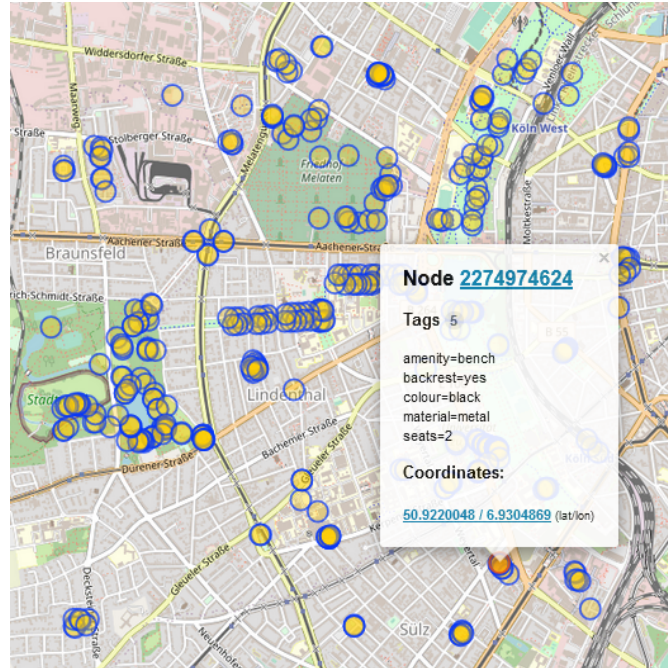


Abbildung 7: Amenities bei OpenStreetMap

Im Anschluss werden die Daten als Geopandas GeoDataFrame gespeichert. Das ist eine tabulare Datenstruktur, welche zusätzlich eine Spalte für geometrische Daten hat.

4.2.1 Problem beim Download

Bei dem Download der Amenities über die Bounding Box kommt es natürlich dazu, dass diese, bei einem von uns generierten Graphen (mit Ausnahmen, wenn die Bounding Box Methode gewählt wurde), nicht genau der Fläche des Graphen entspricht. Das ist ein Problem, da Amenities, die außerhalb der, von dem Graphen eingeschlossenen, Fläche liegen, den Knoten die sich am Rand des Graphens befinden, zugeordnet werden. Dabei werden dementsprechend auch Amenities einem Knoten zugeordnet, obwohl sie eigentlich nicht wirklich in der Nähe dieses Knotens sind. Wie man an folgendem Beispiel erkennen kann, ist das, gerade bei dem Generieren mittels der gewöhnlichen Breitensuche, ein Problem.

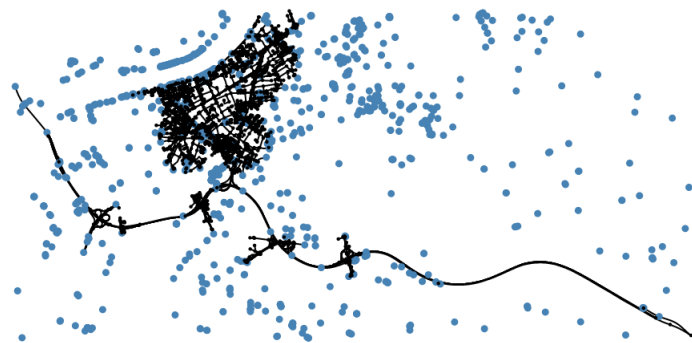


Abbildung 8: Graph mit ungünstiger Verteilung der Amenities

Hier würden den äußeren Knoten des Graphens Amenities zugewiesen werden, welche eigentlich sehr weit entfernt sind. Die Straßen die sehr weit von den anderen Knoten wegführen, vergrößern das von der Bounding Box eingeschlossene Gebiet. Wie man sehen kann, werden von OSMnx mittels der Overpass API, viele Amenities heruntergeladen, welche nicht in dem, vom Graphen eingeschlossenen Gebiet, liegen. Das wird bereits beim Generieren verbessert, aber dennoch bleibt das Problem, auch bei Graphen ohne dieser Art von Straßen, bestehen. Es lässt sich geometrisch leicht erklären, warum das bei einer Box welche, man über einen ungefähr runden Graphen legt, passiert.

Die Lösung für dieses Problem ist jedoch genauso simpel. Es wird einfach beim finden des nächsten Knotens für jede Amenity erst geprüft, ob die Distanz zu diesem Knoten ein wählbares Limit überschreitet. Sollte sie zu weit entfernt sein, wird sie beim Labeln ignoriert. So wird einfach verhindert, dass das Ergebnis verfälscht wird.

4.3 Labeln der Knoten

4.3.1 Euklidische Distanz

Am Anfang der Entwicklung haben wir zunächst nur die euklidische Distanz zum Labeln verwendet. Unser Algorithmus findet für jede Amenity den nächsten Knoten über die euklidische Distanz, und passt den Labelvector dieses Knotens an. Im folgenden Bild wird gezeigt wie gefundene Amenities dem nächsten Knoten gemäß der euklidischen Distanz zugeordnet werden. Die bunten Punkte stellen die Amenities da und der entsprechende Knoten wird immer in die gleiche Farbe eingefärbt. Dabei handelt es sich um ein echtes Straßennetzwerk, auch wenn es sich dabei nur um einen Ausschnitt eines größeren Graphens handelt.

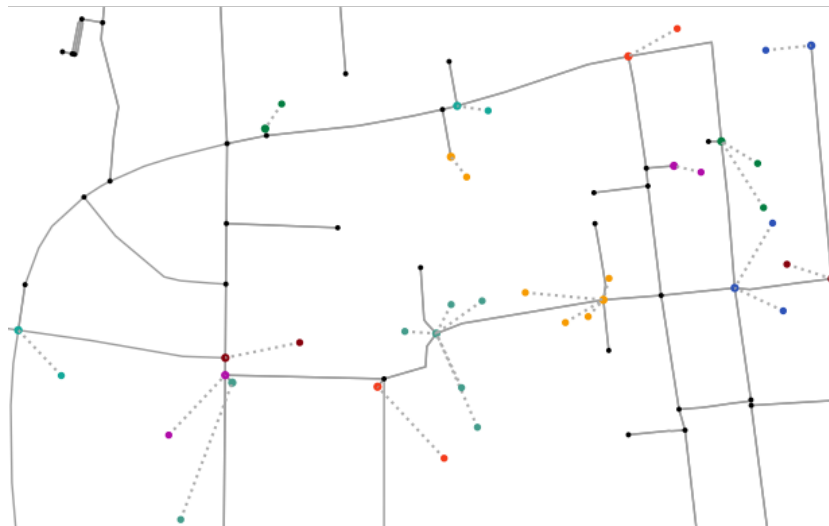


Abbildung 9: Zuweisung über euklidische Distanz

Man erkennt, dass die Zuweisung zuverlässig den nächsten Knoten findet. Im Laufe der Entwicklung haben wir uns jedoch die Frage gestellt, ob diese Art, den nächsten Knoten zu finden, immer der Sinnvollste ist.

4.3.2 Die euklidische Distanz ist nicht immer geeignet

Die Zuweisung über die Euklidische Distanz ergibt nicht, für jede Amenity oder Situation, Sinn. Da es sich bei Kanten um Straßen handelt, ist es sehr gut denkbar, dass man den kürzesten Weg über eine Straße nutzen möchte und nicht die kürzeste Luftlinie. Auch wenn zum Beispiel Häuser den direkten Weg unmöglich machen, ist es sinnvoll sich an den Kanten zu orientieren. In dem folgenden Bild stellen die braunen Rechtecke Häuser da, welche den direkten Weg versperren. Der rote Punkt stellt eine beliebige Amenity da.

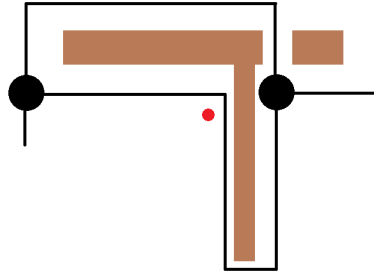


Abbildung 10: Problem bei euklidischer Distanz

Es lässt sich erkennen, dass in so einem Fall die euklidische Distanz offenbar nicht das optimale Maß ist. Wenn man die Amenity dem Rechten Knoten, welcher mittels der euklidischen Distanz der nähere ist, zuweist, wäre der Weg weiter, als wenn man den linken Knoten wählen würde.

4.3.3 Nearest Edge

Um dieses mögliche Problem zu adressieren, haben wir uns eine weitere Methode zum Labeln überlegt. Für jede gefundene Amenity wird dabei der nächste Knoten nicht mehr über die euklidische Distanz bestimmt. Stattdessen wird der nächste Knoten so bestimmt, dass zunächst für jede Amenity die nächste Kante berechnet wird. Sobald die nächste Kante gefunden wurde, wird der Punkt auf der Kante bestimmt, von dem aus der Weg zur Amenity am kürzesten ist. Danach wird die Distanz von diesem Punkt zu den beiden inzidenten Knoten berechnet. Die Amenity wird dann dem Knoten der näher ist zugeordnet und der Knoten wird entsprechend gelabelt. In folgender Grafik ist auf der Rechten Seite erkennbar, wie eine solche Zuweisung dann, in einem richtigen Straßennetzwerk, aussehen würde. Die Kanten werden so mit der Farbe der Amenity eingefärbt, dass erkennbar ist, welcher Knoten näher ist.

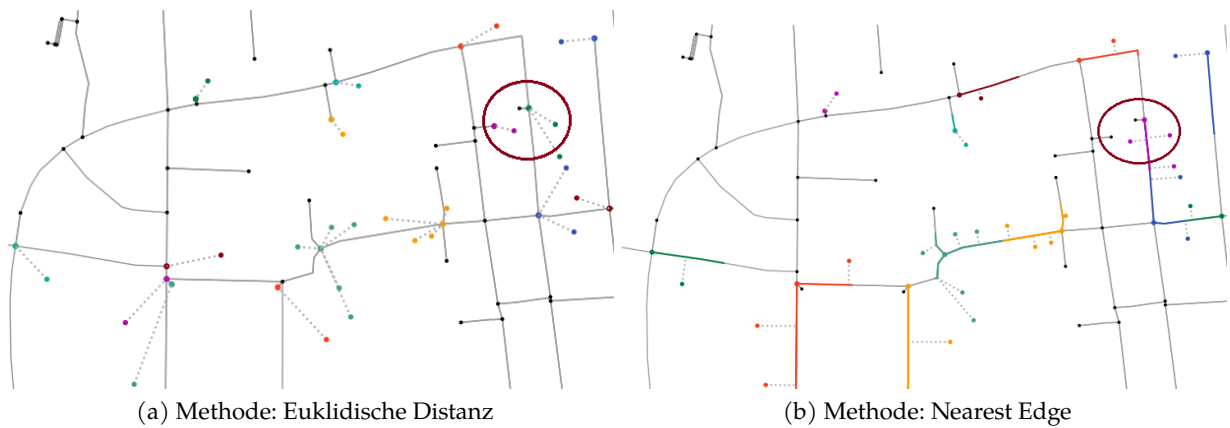


Abbildung 11: Vergleich der beiden Methoden

Wie man in dem markierten Bereich sehen kann, macht das auch in einem realen Beispiel, und nicht nur in unserer Konstruktion, einen Unterschied. Wenn man genau hinschaut kommt es häufiger vor, dass ein anderer Knoten gewählt wird. Auch zum Beispiel wenn eine Straße in einer Sackgasse endet, ist dieser Knoten häufig über Nearest Edge nicht der nächste, bei der euklidischen Distanz hingegen schon.

Dem Nutzer wird überlassen, welche der beiden Methoden für seinen Anwendungsfall die geeignetere ist. Das ist davon abhängig, um was für Amenities es sich handelt, und ob man auf die Straßen limitiert ist.

Abschließend hat man dementsprechend die Wahl zwischen, den sich ausschließenden Modi Binary oder Counter sowie euclidian distance oder nearest edge.

5 Speichern der Graphen

Das Programm stellt drei verschiedene Formate zum Speichern der Graphen zur Verfügung. Diese sind Pickle, GraphML und das selbstentwickelte ETGF Format. Im Folgenden werden die drei Formate zunächst vorgestellt und anschließend miteinander in Bezug auf Geschwindigkeit und Speicherplatzbedarf verglichen.

Robert

5.1 Pickle

Das Python-Modul pickle implementiert binäre Protokolle zum Serialisieren und Deserialisieren von beliebigen Python Objekten. Dementsprechend ist das gleichnamige Format ein Binärformat. Das Serialisieren eines Python Objektes nennt sich „pickling“ und das Deserialisieren „unpickling“. Aufgrund der Tatsache, dass es sich um ein binäres Format handelt, ist das Lesen und Schreiben der Graphen, wie der Vergleich zeigen wird, schneller als bei den beiden textbasierten Alternativen. Das Problem am Pickle Format ist, dass es nicht sicher ist. In der offiziellen Python Dokumentation wird ausdrücklich darauf hingewiesen, dass nur Daten aus vertrauten Quellen deserialisiert werden sollten [pickle — Python object serialization 2022](#). Grund dafür ist, dass die Daten vor dem Einlesen so verändert werden können, dass beim „unpickling“ beliebiger Python Code ausgeführt werden kann. Slaviero, [2011](#) gibt eine detaillierte Einführung in Code Injection Angriffe durch Python Pickle.

Es stellt sich nun die Frage, warum das Pickle Format trotz der Sicherheitsbedenken im Programm verwendet wird. Der erste Grund ist die Performanz, wie im letzten Abschnitt gezeigt wird. Desweiteren appellieren wir an den Nutzer, dass er die Warnung ernst nimmt und wirklich nur Dateien aus vertrauten Quellen einliest. Es kann argumentiert werden, dass die Dateien ohne das Wissen des Nutzers lokal auf dem Rechner manipuliert werden können. Das stimmt, jedoch besteht so schon der direkte Zugriff auf den Rechner und die Möglichkeit böartigen Code durch einen Code Injection Angriff über Pickle auszuführen stellt das kleinere Problem dar.

5.2 GraphML

GraphML steht für Graph Markup Language und basiert auf dem XML Format. Das Format wurde erstmals in einem Workshop während des im Jahr 2000 stattfindenden „Graph Drawing Symposium“ initiiert [[The GraphML File Format 2022](#)]. Das Hauptziel war es ein standardisiertes Format zum Austausch von Graphen zu schaffen. Vorallem sollte es dabei möglich sein beliebige Graphen wie Beispielsweise ungerichtete-, gerichtete-, Multi- und Hypergraphen sowie die zugehörigen Attribute repräsentieren zu können. Zudem wurde besonderes Augenmerk auf Einfachheit und Lesbarkeit, Erweiterbarkeit bezüglich des zugrunde liegenden Graphenmodells und Robustheit gelegt [Brandes u. a., [2013](#)]. Das Format unterstützt die einfachen Datentypen *boolean*, *int*, *float*, *double* und *string*. Diese Einschränkung lässt sich umgehen indem komplexere Datentypen wie Beispielsweise *list<int>* zunächst in string-Repräsentation gespeichert werden und beim Einlesen wieder in eine Liste von Integern umgewandelt werden. Nachteilig dabei ist, dass extern gespeichert werden muss, welche Attribute besonders behandelt werden müssen.

Appendix A zeigt ein Beispiel für einen Graphen mit zwei Knoten und zwei Kanten im GraphML Format. Ohne auf die genaue Struktur, welche in [GraphML Primer 2022](#) beschrieben wird, einzugehen lässt sich leicht erkennen, dass der Anteil an Informationen die den Graphen beschreiben im Vergleich zu dem durch XML ähnlichen Tags erzeugten Text deutlich geringer ist. Selbst, wenn man von der Deklaration der Attribute in den ersten

Zeilen absieht ist der Unterschied noch deutlich zu erkennen. Folglich führt die in GraphML gegebene Flexibilität zu größeren Dateien. In unserem Anwendungsfall wird diese Flexibilität jedoch nicht unbedingt benötigt was die Vermutung nahe legt, dass die Größe der Dateien stark reduziert werden kann.

5.3 ETGF

Die Motivation hinter der Entwicklung des Extended Trivial Graph Format (kurz: ETGF) war es die Dateigröße zu verringern und den Inhalt möglichst auf die Informationen zu beschränken, welche nötig sind um den Graphen zu beschreiben.

Bevor genauer auf das Extended Trivial Graph Format eingegangen wird, folgt zunächst eine kurze Erklärung des zugrunde liegenden Trivial Graph Format (kurz: TGF).

5.3.1 TGF

Das Trivial Graph Format ist ein einfaches, nicht standardisiertes Format zur textuellen Darstellung von Graphen. Es werden gerichtete- und ungerichtete Graphen mit einem Label pro Knoten und Kante, jedoch nicht für den Graphen an sich, unterstützt. Da das Format nicht standardisiert ist findet man teilweise leicht verschiedene Ausführungen. Die folgende Definition scheint die am häufigsten vorkommende zu sein:

```
file = node_list
      #
      edge_list

node_list = node_item node_list | node_item | []
node_item = node_id [node_name]
edge_list = edge_item edge_list | edge_item
edge_item = source_node_id target_node_id [edge_name]
source_node_id = node_id
target_node_id = node_id
```

Abbildung 12: Aufbau TGF Format (Quelle: [yFiles Java 2.17 Docs TGFIOHandler 2022](#))

Abbildung 13 zeigt ein einfaches Beispiel für einen gerichteten Graphen im TGF Format. Der Graph stellt das Familienverhältnis von Tom, Anna und Bert dar. Die Knotenlabel stellen die Namen und die Kantenlabel stellen die Beziehung zwischen den Personen dar.

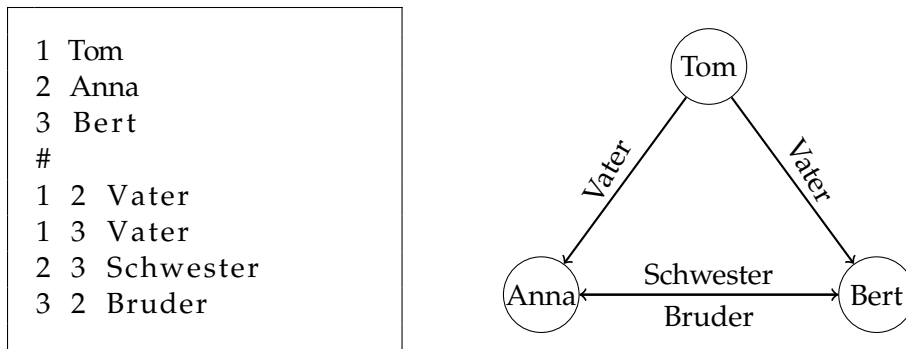


Abbildung 13: Gerichteter Graph in TGF Format

5.3.2 Erklärung ETGF

Das Extended Trivial Graph Format baut auf der Struktur von TGF auf. Es erlaubt jedoch auch Mehrfachkanten sowie multiple Knoten-, Kanten- und zusätzlich auch Graphattribute. Daher auch das „Extended“ im Namen. Umgesetzt wird dies durch eine Kombination des TGF- und dem Comma Separated Value (CSV) Format. Vom Aufbau ähnlich sind das TGF-CSV Format von Spacek, [2022](#) sowie das von GUESS benutzte GDF Format Adar, [2022](#). In beiden Formaten können gerichtete Graphen gespeichert werden. Multikanten lassen sich durch das Attribut „Key“, welches die Multikanten zwischen zwei Knoten indiziert, realisieren. Das Speichern von Graphattributen ist bei beiden Formaten jedoch nicht möglich.

Der allgemeine Aufbau des ETGF Formats ist in Abbildung 14 zu sehen.

```

Datentyp1 , Datentyp2 , ...
Attributname1 , Attributname2 , ...
Graphattribut1 , Graphattribut2 , ...
#
int , Datentyp2 , ...
Knotenid , Attributname1 , ...
Knoten1 , Attribut1 , ...
...
#
int , int , Datentyp3 , ...
u , v , Attributname1 , ...
Knoten_u , Knoten_v , Attribut1 , ...
...

```

Abbildung 14: Aufbau ETGF Format

Der Aufbau des Formats zeichnet sich dadurch aus, dass zuerst die Datentypen der Attributwerte, dann die Attributnamen und zuletzt die Attributwerte angegeben werden. Die

einzelnen Blöcke für den Graphen, die Knoten und die Kanten werden durch ein # getrennt. Das Format erlaubt so zunächst einmal eine einfache Darstellung von gerichteten und Multigraphen. Ungerichtete Graphen lassen sich durch das Angeben der Kante (u,v) und (v,u) durch einen gerichteten Graphen modellieren. Als Attributtypen und Attributwerte ist allgemein alles, in Text darstellbare, erlaubt. Jedoch muss die Implementierung im Programm angepasst werden. Appendix A und Appendix B zeigen den gleichen Graphen im GraphML- sowie im ETGF Format. Es ist offensichtlich, dass das ETGF Format platzsparender ist, was im späteren Vergleich bestätigt wird. Zudem ist der Bezug zu TGF und CSV im ETGF Format gut zu erkennen.

Da unser Programm mit von OSMnx erzeugten Straßennetzwerken arbeitet, sind die erlaubten Datentypen unserem Anwendungsfall entsprechend eingeschränkt. Als Datentypen erlaubt sind die einfachen Datentypen *str*, *int*, *int64*, *float*, *float64*, *bool* sowie *Point* und *Linestring* im Well Know Text Format und CRS in PROJ Format. Außerdem wird der Datentyp *list:<Datentyp>* beziehungsweise *list:na*, wenn der Datentyp nicht bekannt ist erlaubt. Ein besonderer Datentyp ist *mixed:<Datentyp>* welcher angibt, dass ein Attribut als einzelner Wert und als Liste im Graphen vorkommt. Das Speichern und Lesen der Graphen wird in *save_graph.py* implementiert. Die jetzige Implementierung induziert die im Graphen enthaltenen Datentypen beim Schreiben automatisch. Das Schema nach welchem das Induzieren der Datentypen erfolgt ist in Abbildung 15 dargestellt. Beim Lesen werden die Attributwerte, entsprechend der Angabe, in die vorgegebenen Datentypen umgewandelt. Dies ist auch der Grund weshalb die erlaubten Datentypen auf die zuvor genannten eingeschränkt sind. Um diese Einschränkung zu umgehen, könnten beim Einlesen des Graphen die Datentypen und Funktionalität zum Umwandeln der Attributwerte, in den korrekten Datentypen der Funktion übergeben werden. Vergleiche *osmnx.io.save_graphml* für die alternative Implementierung (in GraphML). Wird das Programm jedoch nur über die Benutzeroberfläche verwendet, so reichen die erlaubten Datentypen aus. Es muss beim Einlesen der Graphen darauf geachtet werden, dass folgende Attribute vorhanden sind:

- Graphattribute: *crs* (CRS), *graph_label* (list:str)
- Knotenattribute: *osmid* (int), *y* (float64), *x* (float64), optional: *label_vec* (list:int)
- Kantenattribute: *u* (int), *v* (int), *key* (int), *length* (float)

Die zuvor genannten Attribute müssen unabhängig vom Speicherformat vorhanden sein. Wichtig ist, dass bei den anderen Dateiformaten die Attributwerte nach dem Einlesen der Graphen die oben genannten Datentypen haben. Die restlichen Attribute werden von dem Programm ignoriert, müssen aber weil sie ebenfalls eingelesen werden einen der erlaubten Datentypen haben. Bei Pickle und GraphML können die Datentypen der anderen Attribute beliebig gewählt sein.

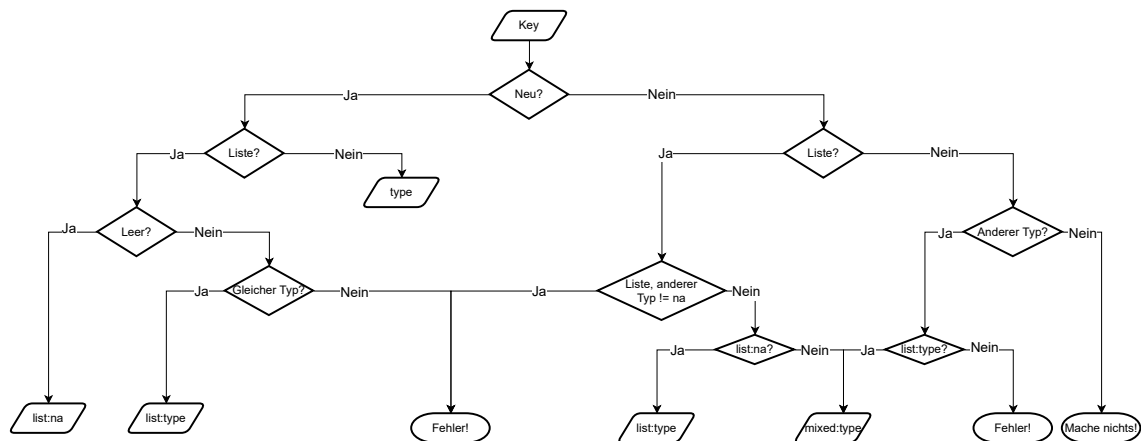


Abbildung 15: Entscheidungsbaum nach welchem die Datentypen induziert werden

5.4 Vergleich von Pickle, GraphML, ETGF

Zuletzt werden die drei Formate bezüglich ihrer Schreib- und Lesegeschwindigkeit sowie ihrer Dateigröße verglichen. Um die Daten zu ermitteln wurde für jedes der Formate jeweils fünf mal gespeichert und gelesen und der Durchschnitt der Werte ermittelt. Es wurde Köln, Deutschland als Ursprungsgraph genommen, aus welchem dann jeweils ein Graph mit 100, 1000, 10000, 100000 Knoten erzeugt wurde. Durchgeführt wurden die Tests auf einem Lenovo Thinkpad T490 mit einem Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 16 GB Arbeitsspeicher und 512GB M.2 PCIe/NVMe (OPAL 2.0) SSD Festplattenspeicher mit Windows 10 als Betriebssystem.

# Knoten	100	1000	10000	100000	# Knoten	100	1000	10000	100000
Pickle	0.005	0.037	0.365	3.795	Pickle	0.002	0.028	0.235	2.768
GraphML	0.027	0.228	2.725	27.464	GraphML	0.028	0.145	1.927	21.231
ETGF	0.009	0.067	0.934	7.353	ETGF	0.012	0.113	1.122	12.819

Tabelle 1: Vergleich Schreiben (links) und Lesen (rechts) in Sekunden

Es fällt wie zuvor schon erwähnt auf, dass Pickle mit Abstand am schnellsten ist. Im Vergleich zu GraphML ist die Schreib- und Lesegeschwindigkeit ungefähr sieben mal schneller, während sich der Unterschied zu ETGF verringert. Lesen ist ungefähr viereinhalb mal schneller und schreiben ungefähr dreimal so schnell. Auch, wenn die Perfomanz des ETGF Format im Schreiben und Lesen nicht so schnell wie das Pickle Format ist, ist es trotzdem noch dreimal schneller im Schreiben und ungefähr eineinhalb mal schneller im Lesen, als das GraphML Format.

# Knoten	100	1000	10000	100000
Pickle	38,1 KB	402 KB	4,02 MB	39,7 MB
GraphML	101 KB	938 KB	9,37 MB	93,8 MB
ETGF	38.1 KB	365 KB	3,69 MB	36,3 MB

Tabelle 2: Vergleich Dateigröße

Den größten Gewinn erzielt das ETGF Format in der Dateigröße. Die erzeugten Dateien sind in den Tests nicht größer als die im Pickle Format und sogar halb so groß wie die Dateien im GraphML Format.

Abschließend stellt sich die Frage, welches Format das „Beste“ ist. Wie sonst auch, ist die Antwort auf diese Frage nicht ganz eindeutig. Offensichtlich ist, dass das GraphML Format am schlechtesten abschneidet dafür aber die höchste Flexibilität bietet. Wenn die Graphen im Nachhinein nicht mehr bearbeitet werden sollen, kann man von diesem Format eher abraten. Wenn die Sicherheit nicht von Interesse ist, kann man das Pickle Format empfehlen. Ein zuvor noch nicht genannter Nachteil ist, dass es nur mit Python kompatibel ist und somit schlechter zum Austausch von Graphen zwischen verschiedenen Programmen geeignet ist. Das ETGF bietet zwar weniger Flexibilität als GraphML überzeugt aber durch den geringeren Speicherbedarf und die erhöhte Geschwindigkeit. Auch zum Austausch zwischen verschiedenen Programmen kann es verwendet werden, weil die Datentypen bis auf *int64* und *float64*, welche durch *int* und *float* ersetzt werden können, nicht direkt von Python abhängig sind. Letztendlich müssen Sie selbst entscheiden, welches Format ihre Anforderungen erfüllt.

6 Statistiken

Nach dem letzten Bearbeitungsschritt im Programm werden zu den Graphen jeweils einzeln, sowie pro Stadt und über alle Graphen zusammen Statistiken berechnet. Diese werden in der Datei *statistics.txt* abgespeichert. Die Statistiken sind im CSV Format, wobei die drei Blöcke durch Überschriften getrennt werden. Es folgt nun eine Auflistung der errechneten Statistiken mit jeweils einer kurzen Beschreibung. Die Kürzel in den Namen der Statistiken stehen für:

Robert

avg Durchschnitt	min Minimum	mode Häufigstes Element
std Standardabweichung	max Maximum	hist Histogramm

Statistiken einzelner Graph:

name Name des Graphen
num_nodes Anzahl Knoten im Graphen
num_edges_loops Anzahl Kanten ohne Mehrfachkanten mit Schleifen
num_edges Anzahl Kanten ohne Mehrfachkanten und Schleifen
num_edges_multi Anzahl Kanten mit Mehrfachkanten und Schleifen
density Dichte des Graphen $num_edges / (num_nodes * (num_nodes - 1))$
out_deg_hist Verteilung der Ausgangsgrade - ohne Mehrfachkanten
avg_deg Durchschnittlicher Ausgangsgrad - ohne Mehrfachkanten
std_deg Standardabweichung Ausgangsgrad - ohne Mehrfachkanten
max_deg Maximaler Ausgangsgrad - ohne Mehrfachkanten
min_deg Minimaler Ausgangsgrad - ohne Mehrfachkanten
mode_deg Häufigster Ausgangsgrad - ohne Mehrfachkanten
out_deg_hist_multi Verteilung der Ausgangsgrade - mit Mehrfachkanten
avg_deg_multi Durchschnittlicher Ausgangsgrad - mit Mehrfachkanten
std_deg_multi Standardabweichung Ausgangsgrad - mit Mehrfachkanten
max_deg_multi Maximaler Ausgangsgrad - mit Mehrfachkanten

min_deg_multi Minimaler Ausgangsgrad - mit Mehrfachkanten
mode_deg_multi Häufigster Ausgangsgrad - mit Mehrfachkanten
label_hist Häufigkeitsverteilung der Einträge in den Labelvektoren pro Amenity
avg_lbl Durchschnitt pro Amenity
std_lbl Standardabweichung pro Amenity
max_lbl Größter Labelvektoreintrag pro Amenity
min_lbl Kleinster Labelvektoreintrag pro Amenity
mode_lb Am häufigsten vorkommender Eintrag pro Amenity
total_edge_length Gesamtkantenlänge - ohne Mehrfachkanten
total_edge_length_multi Gesamtkantenlänge - mit Mehrfachkanten
total_edge_length_by_type Gesamtkantenlänge nach Typ - mit Mehrfachkanten
graph_label Label Attribut des Graphen

Zu den Statistiken die über alle Graphen einer Stadt beziehungsweise über alle generierten Graphen berechnet werden, sind nur die Statistiken die neu hinzukommen oder anders als erwartet berechnet werden angegeben. Bei allen anderen Statistiken wurden jeweils die Einträge aufsummiert und darüber dann die Statistiken berechnet. Zudem können sich die Namen geringfügig unterscheiden.

Statistiken über mehrere Graphen

name Name des Eintrags
sum_num_nodes Anzahl Knoten insgesamt
avg_num_nodes Durchschnittliche Knotenanzahl
std_num_nodes Standardabweichung der Knotenanzahl
sum_num_edges_loops, avg_num_edges_loops, std_num_edges_loops
sum_num_edges, avg_num_edges, std_num_edges
sum_num_edges_multi, avg_num_edges_multi, std_num_edges_multi
density $\text{sum_num_edges} / (\text{sum_num_nodes} * (\text{sum_num_nodes} - 1))$ (insgesamt)
avg_density Durchschnittliche Knotendichte (über alle einzelnen Graphen)
std_density Standardabweichung der Knotendichte (über alle einzelnen Graphen)
sum_lbl Anzahl Label pro Amenity
sum_total_edge_length Gesamtkantenlänge - ohne Mehrfachkanten
avg_total_edge_length Durchschnittliche Gesamtkantenlänge - ohne Mehrfachkanten
std_total_edge_length Standardabweichung Gesamtkantenlänge - ohne Mehrfachkanten
sum_total_edge_length_multi, avg_total_edge_length_multi, std_total_edge_length_multi
sum_total_edge_length_by_type

7 Das Programm

In diesem Kapitel wollen wir einen Überblick über die Funktionen und die Bedienung unseres Programmes geben.

7.1 Die Benutzeroberfläche

Für die Benutzeroberfläche (GUI) haben wir das Window-Toolkit „QT“ benutzt, oder genauer, dessen Python-Port „Pyside 2“. Mithilfe des Programmes „QT Designer“ ist es möglich, per Drag und Drop solche GUIs zu erstellen.

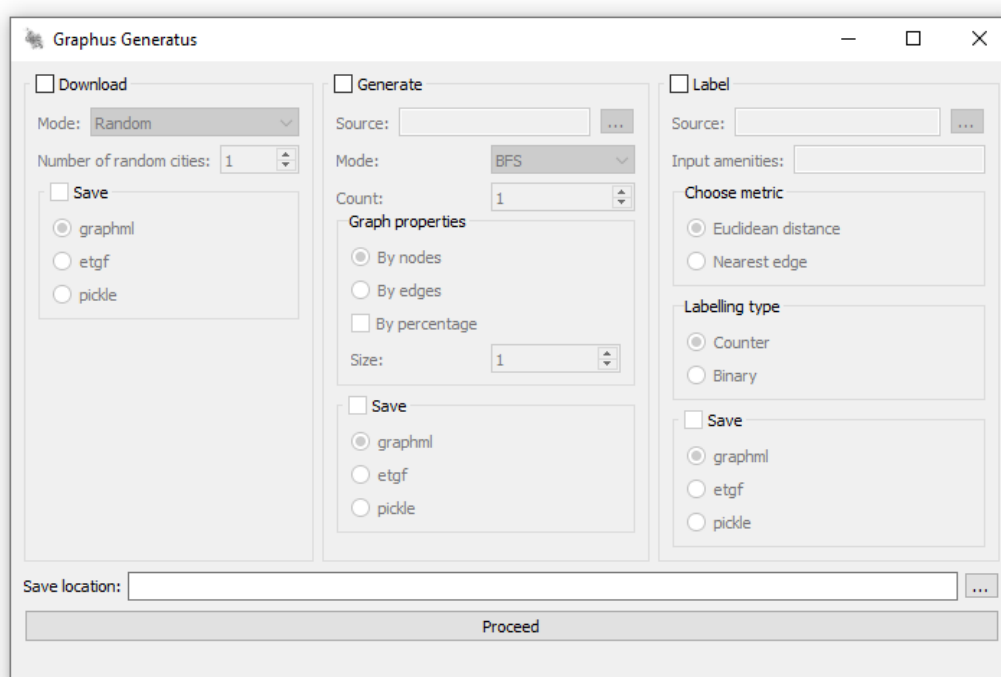


Abbildung 16: Die fertige Benutzeroberfläche unseres Programmes.

7.2 Programmkonfiguration mittels beliebiger Kombination aus den Grundfunktionen Download, Generate und Label

Es wurde bereits erläutert, wie Städtegraphen mittels OSMnx heruntergeladen werden können, und wie aus heruntergeladenen Graphen neue erzeugt und letztlich gelabelt werden. Die drei Grundfunktionen unseres Programmes sind also das Herunterladen von Graphen, das Generieren neuer Graphen und das Labeln von Graphen. Bisher sind wir davon ausgegangen, dass immer jede dieser Grundfunktionen in genau dieser Reihenfolge von unserem Programm ausgeführt wird. Je nach Anwendungsfall sind jedoch nicht immer alle dieser Funktionen notwendigerweise durchzuführen. Beispielsweise kann es sein, dass ein Nutzer sich nur für die generierten Graphen interessiert und sie gar nicht

labeln will. Außerdem könnte es sein, dass ein Benutzer bereits viele verschiedene Städtegraphen heruntergeladen hat, und nur neue Graphen erzeugen bzw. Labeln will. In diesem Fall wäre es von großem Nutzen, wenn die Städtegraphen nicht erneut heruntergeladen werden müssen und stattdessen die bereits heruntergeladenen Graphen stattdessen verwendet werden können. Diese Beispiele verdeutlichen, dass ein flexibles Design unseres Programmes von Vorteil wäre, bei dem jede der drei Grundfunktionen unabhängig voneinander ist. Deshalb haben wir ein Programm entworfen, welches jede Kombination der drei Grundfunktionen „Download“, „Generate“ und „Label“ unterstützt. Im Folgenden wird die Funktionsweise der einzelnen Konfigurationen genauer erläutert.

Download Lädt zufällige oder vom Benutzer angegebene Städtegraphen herunter.

Generate Eine Menge von Graphdateien wird vom Programm eingelesen und aus den eingelesenen Graphen wird jeweils eine festgelegte neue Anzahl an Graphen erzeugt.

Label Eine Menge von Graphdateien wird vom Programm eingelesen, die eingelesenen Graphen werden gelabelt.

Download→Generate Lädt Graphen herunter und labelt dannach die heruntergeladenen Graphen.

Generate→Label Eine Menge von Graphdateien wird vom Programm eingelesen und aus den eingelesenen Graphen wird jeweils eine festgelegte Anzahl neuer Graphen generiert und gelabelt.

Download→Generate→Label Graphen werden heruntergeladen, aus diesen werden neue Graphen generiert und gelabelt.

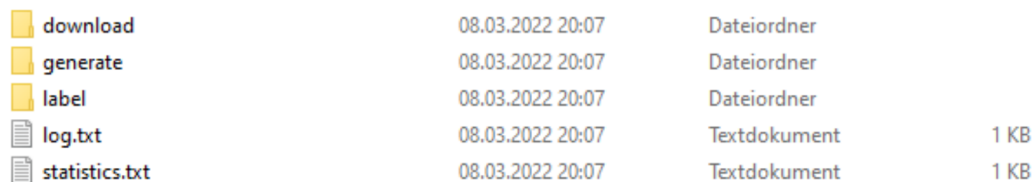
Für die Konfigurationen Generate, Label und Generate→Label muss jeweils ein Ordner spezifiziert werden, an denen die Graphdateien liegen. Die Dateien müssen in einem der drei von uns vorgestellten Formate vorliegen. Die einzelnen Funktionen kann der Benutzer in der oberen Zeile der GUI (Abbildung 16) durch Anklicken der jeweiligen Boxen mit der Beschriftung Download, Generate oder Label aktivieren. Dadurch wird die gesamte Spalte der jeweiligen Funktion zur Einstellung der spezifischen Parameter freigeschaltet.

7.3 Programmeinstellungen

Jede der drei Grundfunktionen bietet verschiedene Modi. Für das Herunterladen von Graphen können entweder eine benutzerdefinierte Anzahl an zufälligen Graphen ausgewählt werden, oder es kann eine Liste von Städten in einem Textfeld oder in Form einer Datei angegeben werden. Die Modi für das Generieren wurden bereits in Abschnitt 3 besprochen. Je nach gewählten Modus verändern sich die Parameter, welche in der GUI spezifiziert werden müssen. Für die Breitensuche bzw. die kompakte Breitensuche muss hier eine maximale Knoten- oder Kantenanzahl (absolut oder in Prozent) angegeben werden. Für Radius und Bounding Box sollte jeweils der Radius oder die Breite und Höhe der Bounding Box angegeben werden. Die Auswahlmöglichkeiten für das Labeln wurden in Kapitel 4 vorgestellt, es ist möglich zwischen Euklidischer Distanz und „Nearest-Edge“ zu wählen und zwischen binärem und nicht-binärem Labeln zu wählen.

7.4 Speicherstruktur

Nach jedem Schritt (Download, Generate, Label) können die Graphen optional in einem der drei in Abschnitt 5 vorgestellten Formate gespeichert werden. Der Speicherort ist für alle drei Grundfunktionen derselbe und kann im Textfeld „Save Location“ angegeben werden. Innerhalb der Save Location wird ein neuer Ordner erstellt, dessen Name mittels Datum und aktueller Uhrzeit versehen wird, um ihn von anderen erstellten Ordnern auseinanderhalten zu können. Während das Programm läuft werden dann, falls die zugehörige Grundfunktion ausgewählt wurde und gespeichert werden soll, die Ordner „download“, „generate“ und „label“ erstellt.



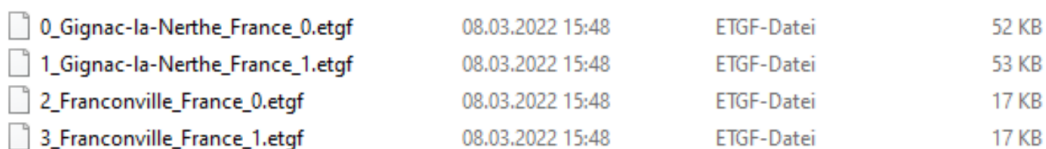
download	08.03.2022 20:07	Dateiordner	
generate	08.03.2022 20:07	Dateiordner	
label	08.03.2022 20:07	Dateiordner	
log.txt	08.03.2022 20:07	Textdokument	1 KB
statistics.txt	08.03.2022 20:07	Textdokument	1 KB

Abbildung 17: Beispiel für den Inhalt des am Speicherort neu erstellten Ordners

Die erstellten Ordner enthalten dann die abgespeicherten Graphen. Die Dateien werden nach folgenden Regeln benannt:

- Heruntergeladene Graphen werden nach dem Schema *STADT_LAND* benannt.
- Leerzeichen werden durch Bindestriche ersetzt.
- Falls neue Graphen erzeugt wurden, wird das Schema um einen Präfix und einen Suffix erweitert: *PRÄFIX_STADT_LAND_SUFFIX*
- Der Präfix ist eine Zahl und sagt aus, der Wievielte der erzeugten Graphen der betrachtete Graph ist.
- Der Suffix ist ebenfalls eine Zahl und sagt aus, der Wievielte der aus einem bestimmten Ursprungsgraphen erzeugten Graphen der betrachtete Graph ist.

Generiert und speichert man beispielsweise jeweils zwei Graphen aus den französischen Städten „Gignac la Nerthe“ und „Franconville“, so sieht der Inhalt des generate-Ordners so aus:



0_Gignac-la-Nerthe_France_0.etgf	08.03.2022 15:48	ETGF-Datei	52 KB
1_Gignac-la-Nerthe_France_1.etgf	08.03.2022 15:48	ETGF-Datei	53 KB
2_Franconville_France_0.etgf	08.03.2022 15:48	ETGF-Datei	17 KB
3_Franconville_France_1.etgf	08.03.2022 15:48	ETGF-Datei	17 KB

Abbildung 18: Beispiel für den Inhalt des generate-Ordners

Außerdem werden noch zwei Textdateien „log.txt“ und „statistics.txt“ erstellt (siehe Abb. 18). *statistics.txt* enthält die gespeicherten Statistiken (vgl. Abschnitt 6). *log.txt* speichert Fehlermeldungen, welche zu einem Programmabbruch führen.

7.5 Fehlerbehandlung und Robusheit

Wir haben uns einige Gedanken gemacht, wie wir unser Programm robuster machen und mögliche Ursachen für einen Programmabsturz beheben können.

Um das Programm, welches mit den Graphen arbeitet von unserer Benutzeroberfläche zu entkoppeln, haben wir die Funktionen für das Herunterladen, Generieren und Speichern auf einen neuen Thread ausgelagert. Außerdem wird eine auf diesem Thread entstandene, nicht behandelte Exception abgefangen. In diesem Fall wird zwar der aktuelle Thread terminiert, der Main-Thread läuft jedoch weiter. Dadurch wird ein Absturz des gesamten Programmes verhindert und die GUI kann weiterhin verwendet werden.

7.5.1 Eingabevalidierung

Bevor das Programm gestartet wird, werden zunächst alle Eingaben auf Ihre Zulässigkeit geprüft. Abhängig von den ausgewählten Grundfunktionen müssen verschiedene Felder ausgefüllt sein. Außerdem müssen die Textfelder, in welchen die zu heruntergeladenen Städte und die Amenities angegeben werden, das richtige Format haben. Dies wird durch eine Überprüfung mittels Regex-Ausdrücken realisiert. Einzelne Stadtangaben bestehen aus einem Stadtnamen und einem Landnamen, getrennt von einem Komma. Verschiedene Städte können mit einem Semikolon voneinander getrennt werden. Amenities im „Input Amenities“-Textfeld werden jeweils mit einem Komma getrennt. Bei fehlenden oder falschen Eingaben wird der Nutzer auf die Fehlerquelle hingewiesen.

7.5.2 Häufige Fehlerquellen und Logging

Mögliche Fehler, welche uns beim Testen des Programmes aufgefallen sind, lassen sich meistens auf das Einlesen bzw. Schreiben von Graphdateien bzw. das Berechnen der Statistiken zurückführen. Beispielsweise kann es passieren, dass eine eingelesene Graphdatei nicht das vorgesehene Format hat. Diese Fehlermeldungen werden in die bereits in 7.4 erwähnte Log-Datei geschrieben. In eine solche Log-Datei könnten gegebenenfalls noch weitere Auskünfte über den Programmablauf ausgegeben werden, dies haben wir allerdings nicht implementiert.

7.6 Performanzanalyse anhand einer Benchmark

Um einen groben Überblick über die Laufzeiten einzelner Programmteile zu erhalten, haben wir ein Benchmark-Programm erstellt. Dieses Programm lädt zuerst den gesamten Graphen der Stadt Köln herunter. Dann werden 50 neue Graphen mit einem Knotenanteil von 30 Prozent mittels normaler Breitensuche generiert und anschließend gelabelt. Beim Labeln werden die euklidische Distanz und nicht-binäre Labelvektoren verwendet. Schließlich werden die Graphen im Pickle-Format abgespeichert. Es werden nur die gelabelten Graphen gespeichert. Das folgende Diagramm 19 zeigt den Anteil der einzelnen Programmteile in Prozent.

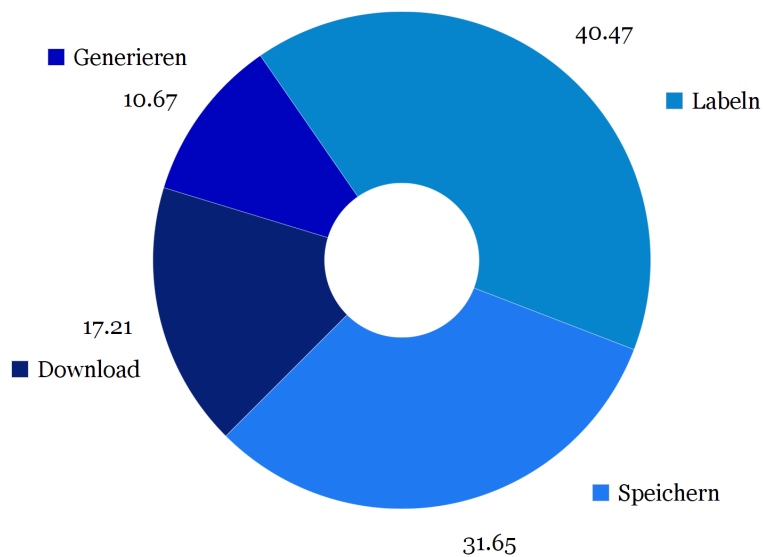


Abbildung 19: Ergebnisse des Benchmark-Tests (Angaben in Prozent der Gesamtlaufzeit)

Auffällig ist, dass das Generieren mit ungefähr 10 Prozent der Gesamtzeit die kürzeste Laufzeit benötigt. Dies erklärt auch, warum die in Abschnitt ?? verglichenen Laufzeiten der verschiedenen Modi keinen großen Einfluss auf die Gesamtlaufzeit des Programmes haben.

Das Herunterladen der Graphen großer Städte dauert zwar sehr lange, da der Graph allerdings nur einmal heruntergeladen wird, nimmt das Herunterladen in diesem Benchmark-Programm keinen großen Laufzeitanteil ein. Am Längsten dauern stattdessen das Speichern und das Labeln der Graphen. Zusammen nehmen diese beiden Programmteile mehr als 70 Prozent der gesamten Laufzeit ein.

Je nachdem, wie das Programm konfiguriert wurde, variieren die aufgezeigten Verhältnisse jedoch deutlich. Zum Beispiel würde die Nutzung des GraphML-Formats die benötigte Laufzeit des Speichern vervielfachen. Dies würde den Anteil des Speicherns wesentlich erhöhen.

8 Abschließende Bemerkungen

Abschließend wollen wir die Ergebnisse unserer Projektgruppe zusammenfassen und Ausblicke für mögliche weitere Entwicklungen des Programmes geben. Wie in den Anforderungen in der Einleitung gefordert, kann unser Programm, gesteuert über eine Benutzeroberfläche, Straßennetzwerke mittels OSMnx herunterladen. Daraus können Teilgraphen generiert und gelabelt werden. Diese drei Funktionen kann unser Programm in einer beliebigen Kombination ausführen. Bei jedem dieser Schritte können die Straßennetzwerke als Graphen gespeichert werden. Um das Speichern der Graphen zu verbessern haben wir das Extended Trivial Graph Format (kurz ETGF) entwickelt. Dieses Format dominiert das GraphML-Format in allen Punkten und verbraucht nicht mehr Speicherplatz als das binäre Pickle-Format. Lediglich bei der Lese- und Schreibgeschwindigkeit kann ETGF nicht mit Pickle mithalten.

Des Weiteren existieren verschiedene Modi für das Generieren von Graphen, aus denen je

Robert,
Linus,
Luca

nach Anwendungszweck ausgewählt werden kann. Auch beim Labeln besteht die Möglichkeit, aus verschiedenen Modi zu wählen. Das Programm kann bezüglich der Fehlerbehandlung, welche momentan noch rudimentär ist, verbessert werden. Außerdem ist die Wartbarkeit des Programmes vor allem im Programmteil, der den Programmfluss steuert, eingeschränkt. Um eine bessere Wartbarkeit zu erreichen, müssen einige strukturellen Änderungen im Programmcode vorgenommen werden. Trotz dieser Kritikpunkte kann unser Programm in der aktuellen Version zum Generieren von Trainingsdaten genutzt werden.

Literatur

- Adar, Eytan (2022). *GUESS The Graph Exploration System*. URL: http://graphexploration.cond.org/manual.html#_Toc116465166 (besucht am 18.03.2022).
- Brandes, Ulrik u. a. (2013). *Graph markup language (GraphML)*.
- GraphML Primer (2022). graphdrawing.org. URL: <http://graphml.graphdrawing.org/primer/graphml-primer.html> (besucht am 18.03.2022).
- pickle — Python object serialization* (2022). Python version 3.8. Python Software Foundation. URL: <https://docs.python.org/3.8/library/pickle.html> (besucht am 18.03.2022).
- Slaviero, Marco (2011). „Sour Pickles Shellcoding in Python’s serialisation format“. In.
- Spacek, Michal Joseph (2022). *Graph-Reader-TGF-CSV*. URL: <https://github.com/michal-josef-spacek/Graph-Reader-TGF-CSV> (besucht am 18.03.2022).
- The GraphML File Format* (2022). See section: Background. graphdrawing.org. URL: <http://graphml.graphdrawing.org/> (besucht am 18.03.2022).
- yFiles Java 2.17 Docs TGFIOHandler* (2022). Aus Abschnitt: Remarks. yWorks. URL: <https://docs.yworks.com/yfiles/doc/api-json/#/api/y.io.TGFIOHandler> (besucht am 18.03.2022).

A Graph $|V|=2$, $|E|=2$, GraphML

```
<?xml version='1.0' encoding='utf-8'?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="d15" for="edge" attr.name="length" attr.type="string" />
  <key id="d14" for="edge" attr.name="oneway" attr.type="string" />
  <key id="d13" for="edge" attr.name="highway" attr.type="string" />
  <key id="d12" for="edge" attr.name="osmid" attr.type="string" />
  <key id="d11" for="node" attr.name="label_vec" attr.type="string" />
  <key id="d10" for="node" attr.name="lat" attr.type="string" />
  <key id="d9" for="node" attr.name="lon" attr.type="string" />
  <key id="d8" for="node" attr.name="street_count" attr.type="string" />
  <key id="d7" for="node" attr.name="x" attr.type="string" />
  <key id="d6" for="node" attr.name="y" attr.type="string" />
  <key id="d4" for="graph" attr.name="graph_label" attr.type="string" />
  <key id="d3" for="graph" attr.name="simplified" attr.type="string" />
  <key id="d2" for="graph" attr.name="crs" attr.type="string" />
  <key id="d1" for="graph" attr.name="created_with" attr.type="string" />
  <key id="d0" for="graph" attr.name="created_date" attr.type="string" />
  <graph edgedefault="directed">
    <node id="4205005751">
      <data key="d6">5660093.237460622</data>
      <data key="d7">362095.70161549986</data>
      <data key="d8">3</data>
      <data key="d9">7.0314115</data>
      <data key="d10">51.0757607</data>
      <data key="d11">[0, 0]</data>
    </node>
    <node id="4205005722">
      <data key="d6">5660062.515746264</data>
      <data key="d7">362106.9297903463</data>
      <data key="d8">3</data>
      <data key="d9">7.0315834</data>
      <data key="d10">51.0754873</data>
      <data key="d11">[0, 0]</data>
    </node>
    <edge source="4205005751" target="4205005722" id="0">
      <data key="d12">420553189</data>
      <data key="d13">footway</data>
      <data key="d14">False</data>
      <data key="d15">32.687</data>
    </edge>
    <edge source="4205005722" target="4205005751" id="0">
      <data key="d12">420553189</data>
      <data key="d13">footway</data>
      <data key="d14">False</data>
      <data key="d15">32.687</data>
    </edge>
    <data key="d0">2022-03-18 23:16:58</data>
    <data key="d1">OSMnx 1.1.1</data>
    <data key="d2">+proj=utm +zone=32 +ellps=WGS84 +datum=WGS84 +units=m +no_defs +type=crs</data>
    <data key="d3">True</data>
    <data key="d4">['bergisch neukirchen', 'germany']</data>
  </graph>
</graphml>
```

Listing 1: Graph im GraphML Format ($|V|=2$, $|E|=2$)

B Graph $|V|=2$, $|E|=2$, ETGF

```
str , str , CRS , bool , list : str , list : float
created_date , created_with , crs , simplified , graph_label , bbox_before_projected
2022-03-18 23:16:58 , OSMnx 1.1.1 , + proj=utm +zone=32 +ellps=WGS84 +datum=WGS84 +units=m +no_defs +type=crs , True ,
    "[ 'bergisch neukirchen' , 'germany' ]" , "[ 51.0754873 , 7.0314115 , 51.0757607 , 7.0315834 ]"
#
int , float64 , float64 , int64 , float64 , float64 , list : int
osmid , y , x , street_count , lon , lat , label_vec
4205005751 , 5660093.237460622 , 362095.70161549986 , 3 , 7.0314115 , 51.0757607 , "[ 0 , 0 ]"
4205005722 , 5660062.515746264 , 362106.9297903463 , 3 , 7.0315834 , 51.0754873 , "[ 0 , 0 ]"
#
int , int , int , int , str , bool , float
u , v , key , osmid , highway , oneway , length
4205005751 , 4205005722 , 0 , 420553189 , footway , False , 32.687
4205005722 , 4205005751 , 0 , 420553189 , footway , False , 32.687
```

Listing 2: Graph im ETGF Format ($|V|=2$, $|E|=2$)