

G01: Kürzeste Wege in Graphen

Boden Luis Andrés
Goldbeck Mattis
Großkloß Oliver
Heidenreich Ben

22. Juni 2025

Inhaltsverzeichnis

1 Einleitung	2
2 Beschreibung und Datenvorbereitung	2
2.1 Graphentheoretisches Problem	2
2.2 Dijkstra-Algorithmus	2
2.3 Datenaufbereitung und Verarbeitung	2
3 Implementierung und Visualisierung	3
3.1 Dijkstra-Algorithmus	3
3.2 Berechnung des kürzesten Pfads	4
3.3 Auffinden der zentralsten Haltestelle	6
3.4 Visuelle Darstellung	6
4 MVV - Netz als Graphen	9
4.1 Modellierung des MVV-Netzes als Graph	10
4.2 Vorgehensweise zur Generierung des Graphen	10
4.3 Datenaufbereitung für Abfahrtszeiten	12
4.4 Dijkstra-Algorithmus mit Abfahrtszeiten	13
4.5 Deployment als containerisierte Web-App	13
5 Fazit	13
6 Ausblick	14

1 Einleitung

Viele Menschen nutzen täglich Navigationsdienste, um Routen zu planen. Alleine Google Maps hat jeden Monat über 2 Milliarden Anwender (Russell 2019). Trotzdem machen sich die wenigsten Gedanken über die zugrunde liegenden Algorithmen. In dieser Arbeit gehen wir auf die Funktionsweise des Dijkstra-Algorithmus ein und wenden diesen auf das Münchener Verkehrsnetz an.

2 Beschreibung und Datenvorbereitung

2.1 Graphentheoretisches Problem

Ziel ist es, den kürzesten Weg innerhalb des Münchener Verkehrs- und Tarifverbund GmbH (MVV) Netzes zu finden. Hierbei werden die Knoten als Haltestellen und die möglichen Verbindungen zwischen den Haltestellen als Kanten interpretiert.

Um den kürzesten Weg ausgehend von einem Punkt zu finden, wird der Dijkstra-Algorithmus genutzt.

2.2 Dijkstra-Algorithmus

Der Algorithmus liefert alle besten Verbindungen, ausgehend von einem beliebigen Startknoten. Für ein nun gewähltes Ziel ermittelt sich der perfekte Weg über die Vorgängerliste. Er setzt sich aus folgenden Schritten zusammen:

- Jeder Knoten bekommt die Eigenschaften Distanz und Vorgänger und wird als unbesucht abgespeichert.
- Die Distanz des Startknotens wird auf 0, alle anderen auf ∞
- Solange unbesuchte Knoten vorhanden sind wird der Knoten mit der geringster Distanz gewählt
- Dieser Knoten wird nun als besucht markiert
- Alle Nachbarn des aktuellen Knoten werden betrachtet und jede Distanz die über den aktuellen Knoten günstiger erreicht werden kann wird aktualisiert

Die Effizienz des Algorithmus beruht auf der gezielten Auswahl des jeweils nächstgelegenen Knotens. Dadurch werden unnötige Berechnungen vermieden und die kürzeste Verbindung besonders schnell gefunden.

2.3 Datenaufbereitung und Verarbeitung

Um den Dijkstra-Algorithmus auf das Münchener Tramnetz anwenden zu können, benötigt es eine geeignete Datenbasis und eine passende Datenstruktur. Das Münchener Tramnetz eignet sich besonders gut, da es ein engmaschiges, gut vernetztes System darstellt und nicht nur aus einfachen Linien oder Baumstrukturen besteht. Ein Verkehrsnetz ist von Natur aus darauf ausgelegt, Verbindungen zwischen verschiedenen Punkten herzustellen,

welches perfekt zu den Anforderungen des Dijkstra-Algorithmus passt. Zudem ist das Netz vielen Menschen im Alltag bekannt, beispielsweise für den Weg zur Hochschule München. Die Implementierung des Dijkstra-Algorithmus im Tramnetz ist unter `python/simple_demo/` zu finden. Die zugrunde liegenden Daten werden in JSON-Dateien gespeichert. Es gibt eine Datei, die alle Haltestellen enthält, sowie eine weitere Datei mit Informationen zu den Tramlinien und deren jeweiligen Start- und Endpunkten. Die gewählte Datenstruktur muss sowohl Kanten als auch Knoten effizient speichern können, ohne die Geschwindigkeit des Algorithmus zu beeinträchtigen. Für den Dijkstra-Algorithmus wird zunächst eine Liste aller Knoten erstellt, gefolgt von einer Zuordnung aller Kanten pro Knoten. Ein Dictionary erweist sich hier als besonders geeignet, da es den Zugriff auf alle Schlüssel (Keys) und die zugehörigen Werte (Values) eines Schlüssels ermöglicht und somit eine schnelle Verarbeitung der Daten unterstützt. Damit die optimierte Form verwendet werden kann, wird ein Parser geschrieben, der JSON-Dateien in ein Python Dictionary umwandelt. Jeder Knoten (also jede Haltestelle) wird als Schlüssel gespeichert, dem als Wert ein weiteres Dictionary zugeordnet ist. Dieses enthält alle direkt verbundenen Nachbarknoten sowie die jeweilige Entfernung zwischen den Haltestellen. Ein Beispiel für einen Eintrag wäre:

```
1 "Hauptbahnhof": {"Stachus": 5, "Sendlinger Tor": 4}
```

Listing 1: Adjazenzliste als Python-Dictionary

Jeder Knoten hat als Value ein Dictionary mit jedem verbundenen Knoten und die entsprechende Distanz.

3 Implementierung und Visualisierung

3.1 Dijkstra-Algorithmus

Im Folgenden wird erläutert, wie der Dijkstra-Algorithmus in Python implementiert wurde, um in einem gewichteten Graphen die kürzesten Pfade von einem Startknoten zu allen anderen Knoten zu finden. Zunächst werden Funktionssignatur und Eingabeparameter beschrieben, gefolgt von einer Darstellung des Ablaufes.

Funktionssignatur Die Implementierung besitzt die folgende Signatur:

```
1 def dijkstra(graph, start_node, weight='weight'):
2     # [...]
3     return distances, predecessors
```

Listing 2: Signatur der `dijkstra`-Funktion

Argumente

- `graph` (dict): Adjazenzliste als Python-Dictionary (siehe Listing 1)
- `start_node`: Der Knoten, von dem aus die Pfade berechnet werden.
- `weight` (str, optional): Name des Schlüssels im Attribut-Dictionary, unter dem das Kantengewicht steht (Standard: `'weight'`).

Ablauf und Implementierungsdetails Zunächst werden alle Distanzen auf ∞ gesetzt, mit Ausnahme des Startknotens, dessen Distanz auf 0 festgelegt wird. Parallel dazu wird ein `predecessors`-Dictionary initialisiert, um später die Vorgänger jedes Knotens zu speichern. Ein Min-Heap (Priority Queue) enthält zu Beginn nur den Startknoten mit Distanz 0. Solange die Warteschlange nicht leer ist, wird der Knoten mit der aktuell geringsten Distanz entnommen.

1. Trägt der aktuelle Knoten nicht zum optimierten Weg bei, wird er übersprungen, oder sonst als "besucht" markiert
2. Für jeden Nachbarn wird geprüft, ob der Weg über den aktuellen Knoten eine geringere Distanz liefert.

$$d(v) = \min(d(v), d(u) + w(u, v))$$

3. Führt dieser Vergleich zu einer Verbesserung, werden `distances[v]` und `predecessors[v]` aktualisiert und der Nachbar mit seiner neuen Distanz erneut in die Priority Queue eingefügt.

Dieser Vorgang wiederholt sich, bis keine Knoten mehr in der Warteschlange verbleiben. Am Ende liefert die Funktion zwei Dictionaries:

- `distances`: Kürzeste Distanzen vom Startknoten zu jedem erreichbaren Knoten.
- `predecessors`: Vorgängerknoten auf dem jeweiligen kürzesten Pfad.

Komplexität Die Zeitkomplexität hängt von der verwendeten Datenstruktur für die Prioritätswarteschlange ab. Bei Verwendung einer einfachen Liste oder eines Arrays beträgt die Komplexität $\Theta(|V|^2)$, da das Finden des Minimums eine lineare Suche erfordert. Für dünn besetzte Graphen kann durch den Einsatz eines Binary Heaps eine deutlich bessere Komplexität von $\Theta((|E| + |V|) \log |V|)$ erreicht werden. Python's `heapq`-Modul implementiert einen effizienten Min-Heap mit $O(\log n)$ Zeitkomplexität für die grundlegenden Operationen `heappush` und `heappop`, was die theoretischen Anforderungen für eine optimale Dijkstra-Implementierung erfüllt (vgl. Cormen u. a. 2009, S. 662).

3.2 Berechnung des kürzesten Pfads

Im Folgenden wird beschrieben, wie aus dem beim Dijkstra-Algorithmus erzeugten Vorgänger Dictionary der exakte Pfad von einem Start- zu einem Endknoten rekonstruiert wird. Dies geschieht über die Funktion `get_shortest_path`, die dabei alle Sonderfälle (nicht erreichbarer Knoten, Start = Ziel) berücksichtigt.

Funktionssignatur Die Funktion besitzt die folgende Signatur:

```
1 def get_shortest_path(predecessors, start_node, end_node):
2     # [...]
3     return path_list_or_empty
```

Listing 3: Signatur der `get_shortest_path`-Funktion

Argumente

- `predecessors`: Dictionary, das für jeden Knoten seinen direkten Vorgänger auf dem kürzesten Pfad enthält (z. B. aus `dijkstra(...)`).
- `start_node`: Der Ausgangsknoten s , von dem aus der Pfad beginnt.
- `end_node`: Der Zielknoten t , bis zu dem der Pfad rekonstruiert werden soll.

Vorgehen und Ablauf Die Funktion initialisiert eine leere Liste `path` und setzt `current_node` auf `end_node`. Anschließend wird in einer Schleife rückwärts über das Vorgänger-Dictionary iteriert:

```
path.append(current_node), current_node ← predecessors[current_node]
```

solange `current_node` existiert und noch nicht der Startknoten erreicht wurde.

- Erreicht die Schleife den `start_node`, wird der Vorgang abgebrochen.
- Ist der `end_node` nicht im Dictionary oder lässt sich der Startknoten nicht erreichen, liefert die Funktion eine leere Liste zurück.
- Im Sonderfall `start_node = end_node` und gültigem Eintrag entsteht eine Ein-Einheitsliste [`start_node`].

Abschließend wird `path[::-1]` zurückgegeben, um den Pfad in der richtigen Reihenfolge von Start nach Ziel zu erhalten.

Beispiel Angenommen, das Vorgänger-Dictionary aus `dijkstra` lautet

```
1 predecessors = {
2     'A': None,
3     'B': 'A',
4     'C': 'A',
5     'D': 'B'
6 }
```

und der Pfad von 'A' nach 'D' ist gesucht.

```
1 path = get_shortest_path(predecessors, start_node='A', end_node='D')
2 print(path) # ['A', 'B', 'D']
```

Hinweise

- Für nicht erreichbare Ziele (`end_node` nicht in `predecessors`) oder fehlenden Zusammenhang wird stets [] zurückgegeben.
- Die Funktion setzt voraus, dass `predecessors` konsistent aus einem vollständigen Dijkstra-Durchlauf vorliegt.

3.3 Auffinden der zentralsten Haltestelle

Der Dijkstra-Algorithmus wird auf alle Haltestellen angewendet. Das Ergebnis ist in der Form eines Dictionarys bzw. einer 2D Matrix mit Knoten und den entsprechenden minimalen Distanzen ausgehend vom Startknoten. Danach werden die Summe der Abstände zu allen anderen Haltestellen berechnet. Diese Summe wird mit dem Ergebnis aller anderen Haltestellen verglichen. Die Haltestelle mit den insgesamt niedrigsten Abständen ist die zentralste des Tramnetzes. Für diese Netz ist das die Haltestelle *Stachus*.

3.4 Visuelle Darstellung

Die Methode `plot_graph` bildet den Kern der entwickelten Visualisierung. Ziel war es, nicht nur die Struktur des MVV-Graphen zu zeigen, sondern auch eine intuitive Interaktion mit den Knoten zu ermöglichen.

Hilfsmethoden Neben `plot_graph` wurden folgende Hilfsmethoden implementiert:

- `create_graph(nodes, edges)`
- `convert_path_to_edges(path)`
- Innerhalb von `plot_graph`: `on_pick(event)`

`create_graph(nodes, edges)` erstellt, wie der Name schon andeutet, einen `NetworkX_Multigraph` aus den Daten der JSON-Files.

Mit `convert_path_to_edges(path)` wird der Pfad, der von `dijkstra.get_shortest_path(...)` zurückgegeben wird, in NetworkX-Kanten konvertiert. Mit dem `predecessors`-Beispiel aus Abschnitt 3.2:

```
1 path = dijkstra.get_shortest_path(predecessors, start_node='A', end_node
= 'D')
2 print(path) # ['A', 'B', 'D']
3 print(convert_path_to_edges(path)) # [(('A', 'B'), ('B', 'D'))]
```

Listing 4: Konvertieren zu Kanten

`on_pick(event)` ist der Event-Handler, der die Interaktion mit den Knoten ermöglicht. So erhält jeder Knoten einen "Knopf", der bei Interaktion diese Methode auslöst. Die Logik dahinter wird im Abschnitt *Dijkstra und Interaktion* erläutert.

Layout Das Münchener Tramnetz eignet sich für die Visualisierung hervorragend, da es vereinfacht und vollständig darstellbar ist. Als Orientierung diente folgendes Abbild:

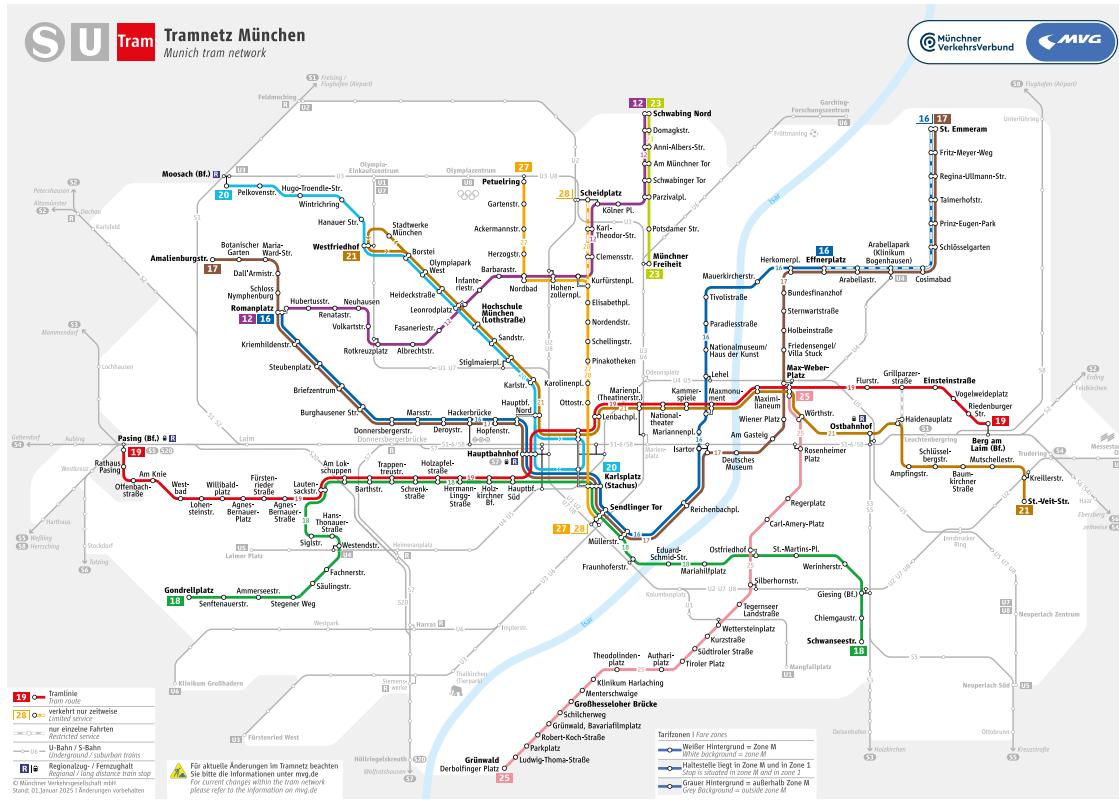


Abbildung 1: MVG Tramnetz (Quelle: Münchener Verkehrsgesellschaft mbH 2025).

Zunächst werden von ChatGPT grob alle Haltestellen mit Namen und Koordinaten extrahiert, was zumindest eine gute Grundlage bietet. Diese werden per Hand korrigiert, damit der Graph entsprechend der Abbildung korrekt aussieht. Es wird NetworkX für das Erstellen des Graphen und matplotlib für das Plotten verwendet. Die Linien werden entsprechend der Abbildung farbkodiert, damit der Verlauf jeder Linie erkennbar ist. Das Ergebnis ist nun mit der Vorlage vergleichbar:

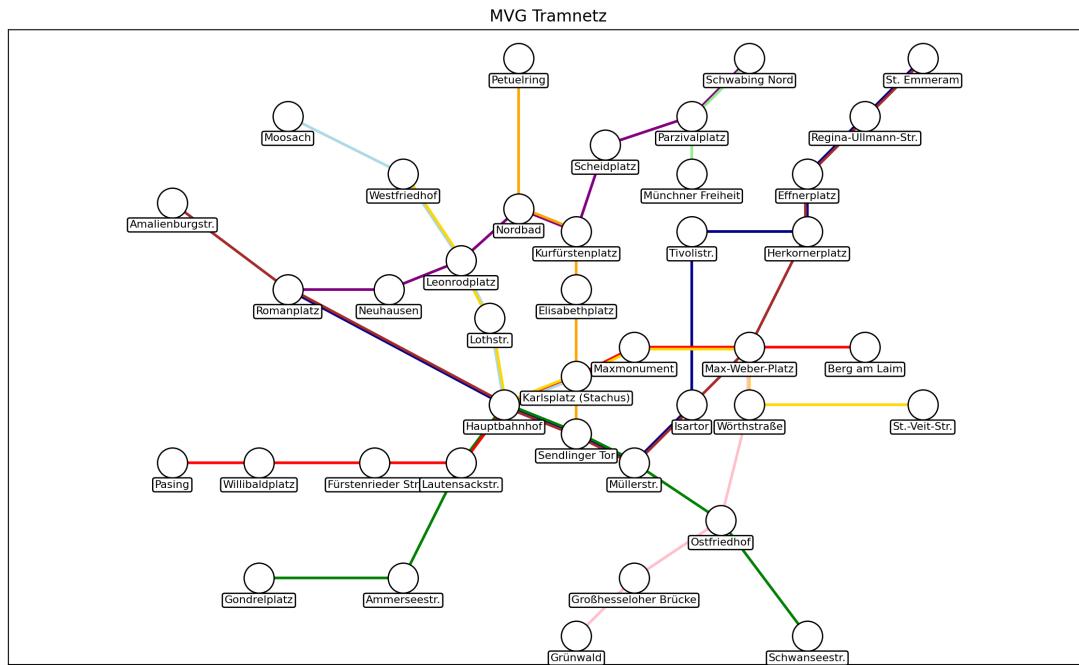


Abbildung 2: Tramnetz als Graph

Dijkstra und Interaktion Um den Dijkstra-Algorithmus zu integrieren, wird der Plot interaktiv gestaltet. Die Idee ist, die Knoten *anklickbar* zu designen, sodass ein Benutzer zwei Knoten auswählen kann. Beim ersten Klick wird Dijkstra von diesem Knoten (markiert durch ein Hexagon) ausgeführt und alle besten Vorgänger gespeichert. Wird ein zweiter Knoten angeklickt, wird die kürzeste Route rot markiert und durch Hexagons an den Haltestellen hervorgehoben. Die Hervorhebungen befinden sich auf einem Layer über dem ursprünglichen Graphen, damit sie bei Bedarf gelöscht werden können, ohne den ursprünglichen Graphen zu verändern.

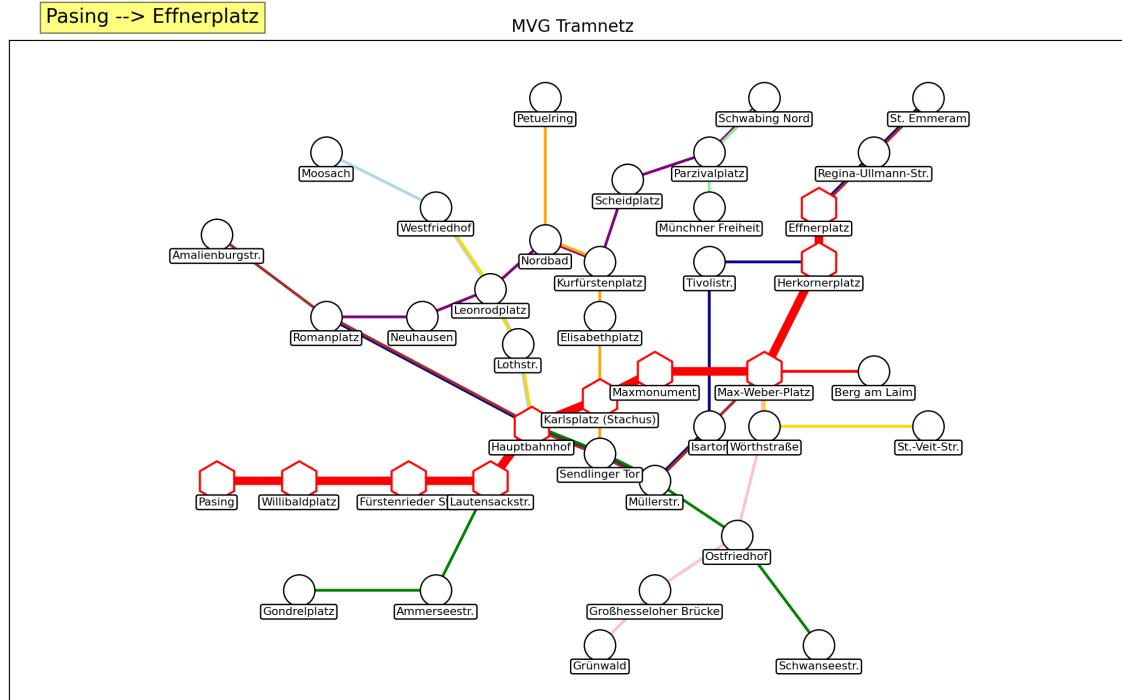


Abbildung 3: Kürzester Pfad von Pasing nach Effnerplatz

4 MVV - Netz als Graphen

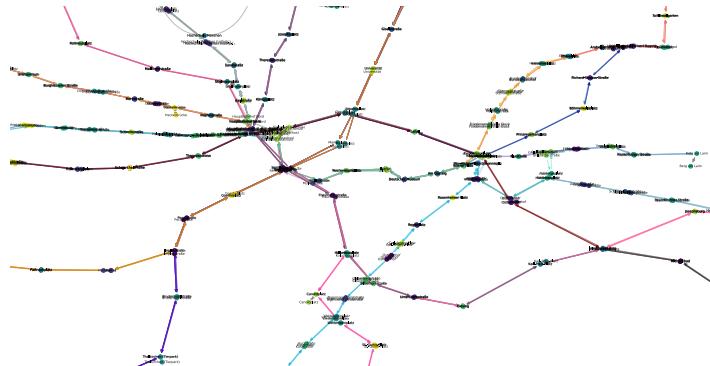


Abbildung 4: Auschnitt aus dem NetworkX Graph Plot des MVV-Netzes mit U-Bahn und Tram

Im Folgenden wird die Erstellung eines NetworkX-Graphen zur Abbildung des MVV-Netzes, U-Bahn und Tram, in München beschrieben. Dieser Graph ist gerichtet, wie bereits in den vorherigen Kapiteln besprochen, und bildet die Grundlage für die Anwendung des Dijkstra-Algorithmus. Diese Erweiterung ist logisch getrennt von der Demo des Tramnetzes und befindet sich im Ordner `python/ptc4gtfs/`. Zunächst werden die Fahrplan-, Haltestellen- und Liniendaten beschaffen. Im nächsten Schritt werden die Rohdaten entsprechend aufbereitet und darauf basierend wird ein MVV-Graph generiert. Abschließend werden die Abfahrtszeiten in die Modellierung mit einbezogen.

4.1 Modellierung des MVV-Netzes als Graph

Die Graph-Modellierung repräsentieren die Knoten der Haltestopp- oder Einstiegsplattformen. Jeder Plattform-Knoten ist über eine Zuordnung mit der übergeordneten Station verbunden, sodass sich das lokale Netz einer Station abbilden lässt. Umsteigekanten setzen wir der Einfachheit halber auf Gewicht 0. Die gerichteten Kanten im Graphen stehen für Transitverbindungen zwischen zwei Plattform-Stops, realisiert durch ein Verkehrsmittel. Wir nutzen dafür einen gerichteten Multigraphen, um unterschiedliche Fahrtrichtungen und Mehrfachverbindungen abzubilden. Dadurch können verschiedene Linien in gleicher oder entgegengesetzter Richtung sowie parallele Verbindungen sauber modelliert werden.

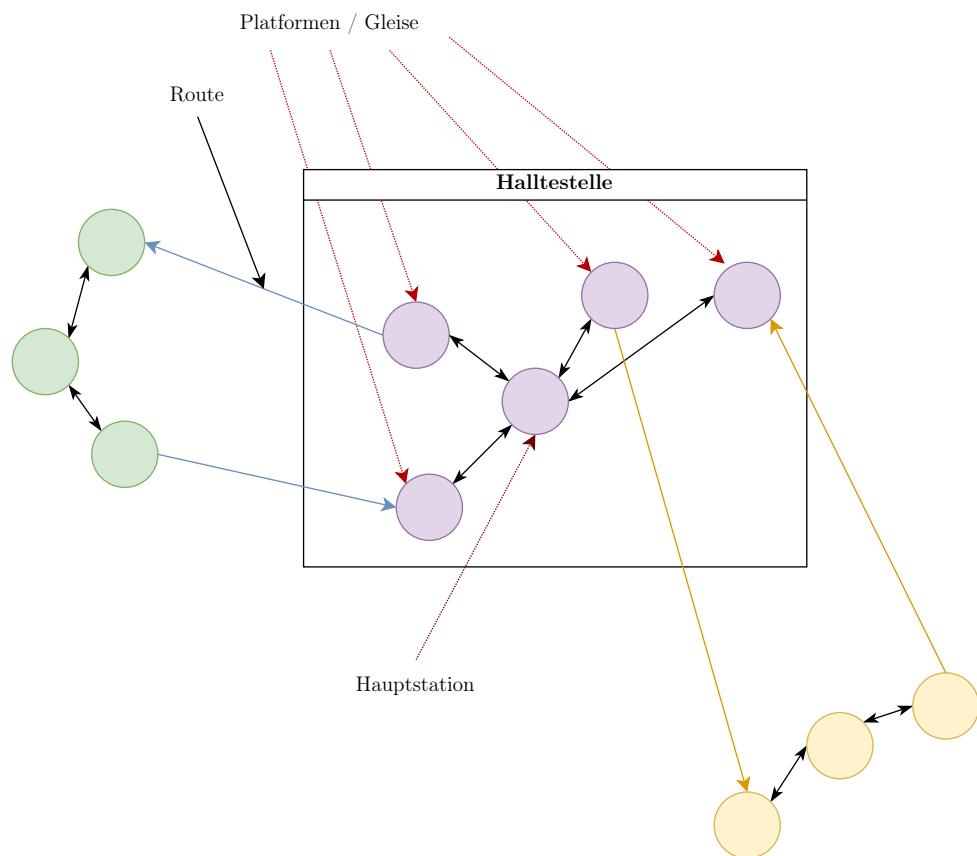


Abbildung 5: Modellierung des MVV-Netzes als Graph

In Abbildung 5 wird dieses Konzept schematisch dargestellt. Dort sieht man, wie Plattformknoten mit Stationen verknüpft sind und wie gerichtete Kanten für die Fahrt zwischen aufeinanderfolgenden Haltestellen angelegt werden.

4.2 Vorgehensweise zur Generierung des Graphen

Um den Graphen mit Echtzeit- bzw. echten Fahrplandaten zu erstellen, nutzen wir den GTFS-Feed für Deutschland von gtfs.de. Diese Feeds werden tagesaktuell bereitgestellt und enthalten Fahrplandaten für Fern-, Regional- und den gesamten Nahverkehr in Deutschland (GTFS.de 2020). GTFS (General Transit Feed Specification) ist ein standardisiertes Datenformat, das Verkehrsbetreibern erlaubt, Service-Daten wie Fahrpläne, Haltestellen

und Routen in mehreren CSV-Dateien bereitzustellen (MobilityData 2025). Die für unsere Zwecke wichtigsten Dateien sind:

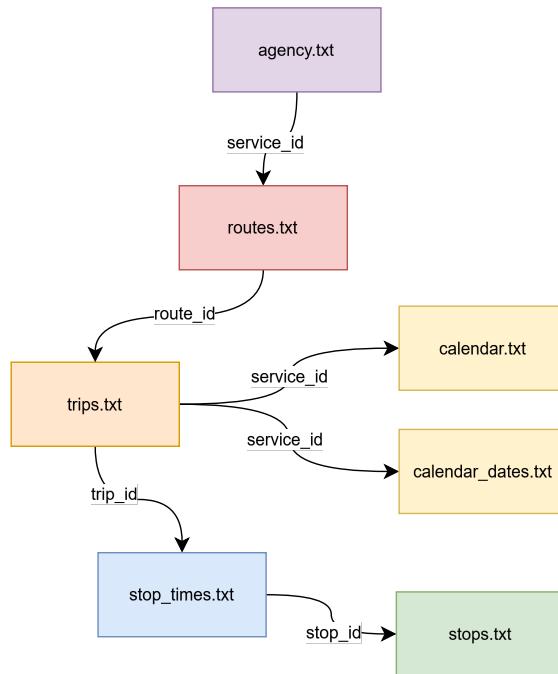


Abbildung 6: GTFS - Zusammenhänge

- **agency.txt:** Informationen zum Verkehrsunternehmen (Name, URL, Sprach- und Zeitzone).
- **stops.txt:** Haltestellen mit eindeutiger ID, Name und geografischen Koordinaten.
- **routes.txt:** Linien mit ID, Typ (Bus, Tram, U-Bahn etc.) und optionaler Farban-gabe.
- **trips.txt:** Einzelne Fahrten auf einer Route, referenziert über `service_id`.
- **stop_times.txt:** Ankunfts- und Abfahrtszeiten jeder Fahrt an den Haltestellen in Reihenfolge.
- **calendar.txt:** Wochentage und Gültigkeitszeitraum (Start-/Enddatum) für jede `service_id`.
- **calendar_dates.txt:** Ausnahmen – zusätzliche (Typ 1) oder ausfallende (Typ 2) Fahrtage.

Der vollständige Deutschland-Feed umfasst große Datenmengen. Daher filtern wir gezielt die Verkehrsmittel in München (MVV) nach U-Bahn sowie Tram. Dazu nutzen wir unser Python-Tool `ptc4gtfs`. Damit werden per `agency_id` alle Datensätze auf den MVV-Anbieter begrenzt. Anschließend legen wir zur effizienten Weiterverarbeitung eine SQLite-Datenbank mit den GTFS Daten an, sodass Abfragen später schneller und strukturierter möglich sind. Für die eigentliche Graph-Erzeugung verwenden wir unser Tool, das

aus der SQLite-Datenbank einen NetworkX-Graphen baut und diesen als serialisiertes Python-Objekt (Pickle) exportiert. Da die Generierung (insbesondere beim Einbeziehen vieler Fahrten und Haltepunkte) aufwändig ist, spart das Nachladen des .pk1-Files in späteren Analyseschritten viel Zeit. In Abbildung 7 ist ein Plot des gesamten U-Bahn-

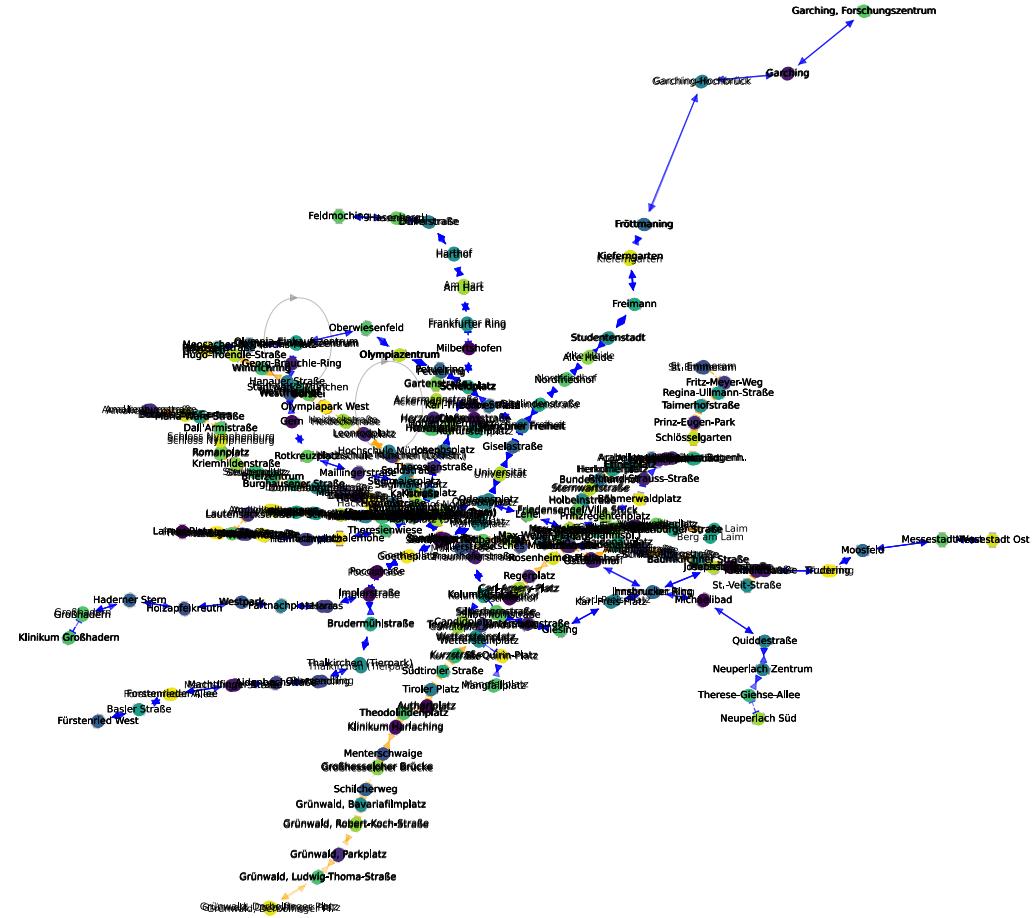


Abbildung 7: NetworkX Graph Plot des MVV-Netzes mit U-Bahn(blau) und Tram(gelb)

/Tram-Teilnetz darstellt. Die Überlagerungen der Label werden durch die hohe Dichte an Haltestellen und Gleisen erzeugt. Selten bilden sich Schleifen, weil manche Stationen im GTFS-Feed ausschließlich als Parent-Stationen vorkommen. Dadurch sind sie sowohl Plattform- als auch Hauptstation und verbinden den Knoten mit sich selbst.

4.3 Datenaufbereitung für Abfahrtszeiten

Um die Abfahrtszeiten effizient im Graphen nutzen zu können, werden die relevanten Informationen aus den GTFS-Tabellen zunächst in einer eigenen Tabelle `departures_today` zusammengefasst. Diese Tabelle enthält für jeden Tag alle gültigen Abfahrten, gefiltert

nach Wochentag, Gültigkeitszeitraum und eventuellen Ausnahmen aus `calendar.txt` und `calendar_dates.txt`. Die Erstellung erfolgt automatisiert über das Python-Tool, das die Datenbank entsprechend vorbereitet. Dadurch lassen sich für jede Haltestelle und Linie die nächsten Abfahrten sehr schnell per SQL abfragen. Das beschleunigt die spätere Routenberechnung erheblich, da keine komplexen Filterungen mehr zur Laufzeit nötig sind.

4.4 Dijkstra-Algorithmus mit Abfahrtszeiten

Für die Routenberechnung im MVV-Netz wurde der klassische Dijkstra-Algorithmus so erweitert, dass er die realen Abfahrtszeiten aus der Tabelle `departures_today` berücksichtigt. Bei jedem Schritt prüft der Algorithmus, wann die nächste passende Abfahrt für die gewünschte Linie an der aktuellen Haltestelle möglich ist, und berechnet daraus die tatsächliche Wartezeit. Diese Wartezeit wird dynamisch zum Kantengewicht addiert, sodass nicht nur die reine Fahrzeit, sondern auch Umstiegs- und Wartezeiten realistisch abgebildet werden. Hierbei werden an manchen Umstiegen noch übermäßig lange Wartezeiten berechnet. Gehwege und Umsteigevorgänge werden als spezielle Kanten mit festen oder 0 Gewichten behandelt. Durch diese Anpassung liefert der Algorithmus realitätsnahe Verbindungen, die den Fahrplanbetrieb im MVV möglichst genau abbilden.

4.5 Deployment als containerisierte Web-App

Für die praktische Nutzung der MVG-Verbindungsabfrage wurde eine Flask-basierte Webanwendung entwickelt, die vollständig containerisiert und einfach deploybar ist. Das Deployment erfolgt über ein kleines Dockerfile, das den gesamten Aufbau automatisiert: Beginnend mit einem Python 3.11 Base-Image wird der Quellcode direkt von GitHub geklont, alle erforderlichen Dependencies installiert und die GTFS-Datenaufbereitung durchgeführt. Die Initialisierungsschritte umfassen das Herunterladen und Filtern der MVV-GTFS-Daten für die Stadtwerke München, die Erstellung der SQLite-Datenbank sowie die Generierung des NetworkX-Graphen für Tram- und U-Bahn-Verbindungen. Nach dem erfolgreichen Build kann der Container mit einem einfachen `docker run`-Befehl gestartet werden und stellt die Webanwendung auf Port 5346 bereit. Diese Architektur ermöglicht eine schnelle und reproduzierbare Bereitstellung der Anwendung in verschiedenen Umgebungen, ohne dass manuelle Konfigurationsschritte oder lokale Datenaufbereitung erforderlich sind. Die Web-App bietet dann eine benutzerfreundliche Oberfläche zur Abfrage optimaler Routen im MVV-Netz unter Berücksichtigung der aktuellen Fahrplandaten.

5 Fazit

Die Implementierung des Dijkstra-Algorithmus erwies sich als anspruchsvoll, trotzdem funktionierte die praktische Anwendung auf das Münchner Tramnetz effizient. Die entwickelte Datenstruktur mit Python-Dictionaries ermöglichte eine schnelle Wegfindung und die visuelle Darstellung bot eine gute Testumgebung für den Algorithmus. Die Erweiterung auf das MVV-Netz brachte jedoch neue Herausforderungen mit sich: Die Verarbeitung großer Datenmengen, die Integration verschiedener Verkehrsmittel sowie die Berücksichtigung von Abfahrtszeiten erforderten erhebliche Anpassungen. Da eine grafische Darstellung des kompletten MVV-Graphen nicht praktikabel war, wurde erfolgreich

eine Flask-Webanwendung als alternative Benutzeroberfläche implementiert.

6 Ausblick

Für die Weiterentwicklung des Systems stehen mehrere konkrete Schritte im Vordergrund. Die Implementierung von Fußwegen zwischen Plattformen derselben Haltestelle würde die Realitätsnähe erheblich verbessern, wobei die bereits vorhandenen geografischen Koordinaten aus dem GTFS-Feed genutzt werden können. Die Integration von Echtzeitdaten für Abfahrtszeiten und Verspätungen und die korrekte Berechnung der Umsteigezeiten ist wichtig, um die praktische Nutzbarkeit zu steigern. Eine Optimierung der Algorithmusperformance für größere Netzwerke wie ganz Deutschland würden das System zu einer vollwertigen Verkehrsplanungsanwendung ausbauen.

Literatur

- Cormen, Thomas H. u. a. (2009). „Single-Source Shortest Paths“. In: *Introduction to Algorithms*. 3. Aufl. Cambridge, Massachusetts; London, England: The MIT Press. Kap. 24, S. 662. ISBN: 978-0-262-03384-8.
- GTFS.de (2020). *GTFS.de: Tagesaktuelle GTFS-Fahrplandaten für Deutschland*. <https://gtfs.de>. Fortlaufend seit 1. Januar 2020; abgerufen am 21. Juni 2025.
- MobilityData (2025). *What is GTFS? – General Transit Feed Specification*. <https://gtfs.org/getting-started/what-is-GTFS/>. Inhalt fortlaufend gepflegt; abgerufen am 21. Juni 2025.
- Münchener Verkehrsgesellschaft mbH (2025). *Tramnetz München*. https://www.mvv-muenchen.de/fileadmin/mediapool/03-Plaene_Bahnhoefe/Netzplaene/Downloads_2025/A4-Tramnetz-2025-Web.pdf. Stand: 01. Januar 2025; Zugriff am 22. Juni 2025.
- Russell, Ethan (2019). *9 things to know about Google’s maps data: Beyond the Map*. <https://mapsplatform.google.com/resources/blog/9-things-know-about-googles-maps-data-beyond-map/>. Zugriff am 22. Juni 2025.