Graph Database Comparison

Marcel Ciesla & Victoria Oberascher

Projektidee

Ziel dieses Projektes war es, einen Vergleich zwischen den graphbasierten Datenbanken **Neo4J** und **TigerGraph** zu schaffen. Im Fokus standen dabei nicht nur die Performances der beiden Datenbanken, sondern auch Aspekte aus Sicht der Software-Entwicklung wie die Verwendbarkeit der Programmierschnittstellen, Graphalgorithmen und Docker Images. Für den Vergleich sollen beide Datenbanken mit demselben Straßenverkehrsnetzwerk befüllt werden. Dafür sollen OpenStreetMap-Daten verarbeitet werden und alle Straßenknoten und Straßensegmente mit den dazugehörigen Informationen wie die Koordinaten, die Länge der Segmente und die erlaubte Höchstgeschwindigkeit extrahiert werden. Anschließend soll sich eine Python-Anwendung über eine Schnittstelle zu diesen beiden Datenbanken verbinden und den kürzesten Weg zwischen zwei unterschiedlichen Punkten im Straßennetzwerk (gegeben durch Koordinaten) berechnen. Um die Bedienung dieser Anwendung etwas intuitiver zu gestalten, wird zusätzlich eine Frontend-App in React entwickelt. Mit Hilfe von dieser soll man den Start- und Zielpunkt auf einer Karte auswählen können, welche an das Backend übergeben werden. Der berechnete Pfad sowie andere Ergebnisse der jeweiligen Datenbanken sollen anschließend auf dieser Karte visualisiert werden.

Data Collection

Zu Beginn des Projektes war es notwendig, Informationen zu den Knoten und Kanten im Straßennetzwerk zu beschaffen. Für diese Aufgabe haben wir ein

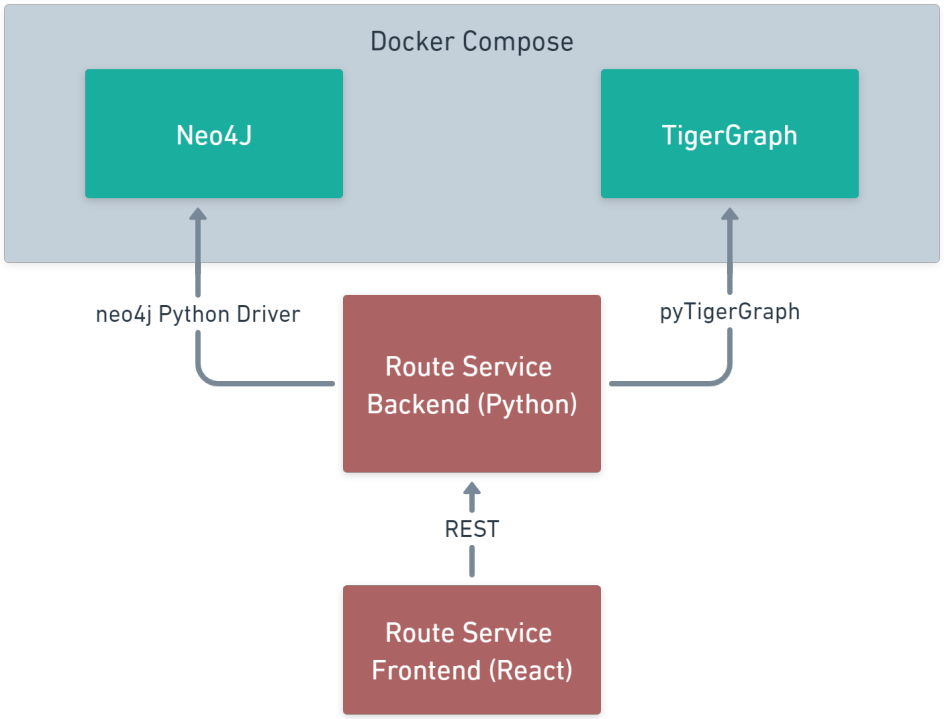
Python-Skript geschrieben, welche mit Hilfe der Library **pyrosm** (https://pyrosm.readthedocs.io/en/latest/) das Straßennetzwerk von einem gegebenen Straßennetzwerk (in unserem Fall **Linz**) herunterlädt. Ursprünglich wäre unsere Idee gewesen, ein größeres Netzwerk zu verwenden (z.B. ganz Österreich). Hier wären die Ladezeiten beim Herunterladen und Verarbeiten für den Rahmen dieses Projektes zu hoch gewesen. Bei dem Netzwerk von Linz hat das Herunterladen selbst schon ca. 3-4 Minuten gedauert. Bei der Library pyrosm kann man einen Parameter **network-type** mitgeben, welcher festlegt, welche Art von OpenStreetMap Daten extrahiert werden sollen. Hier haben wir den Wert "driving" festgelegt, sodass nur tatsächliche Straßen und Kreuzungen verwendet werden (und nicht z.B. Fahrrad- oder Wanderwege).

Die Knoten im Straßennetzwerk haben wir in die Datei **nodes.csv** exportiert. Hier speichern wir die ID und die Koordinaten des Nodes (latitude & longitude). Die Informationen der Kanten werden in die Datei **edges.csv** gespeichert. Hier speichern wir für jede Kante bzw. Straßensegmente die Länge des Segmentes (in Meter) und jeweils eine Referenz (ID) auf den Start- und Endknoten des Segmentes.

Bei der Installation von pyrosm (auf einem Windows-System) haben wir leider einige Probleme gehabt. Da eine Installation unter Linux laut Dokumentation empfohlen wird, haben wir uns dazu entschieden, das Python-Skript in einem eigenen Python Docker Container mit Docker Compose auszuführen. Hier wird zu Beginn das Skript selbst und eine Liste der benötigten Libraries (requirements.txt) auf den Container kopiert. Nachdem die Ausführung des Skripts abgeschlossen ist, werden die finalen CSV-Dateien nodes.csv & edges.csv in den Volume-Ordner results kopiert, welcher anschließend von dem Host-System gelesen werden kann.

Architektur

In der folgenden Abbildung wird die Architektur des Route Service visualisiert. Über den Webbrowser kann der User einen Start- und Zielpunkt auf der Karte auswählen. Das Frontend übergibt diese Parameter mittels einer REST-Schnittstelle an das Backend. Dieses verarbeitet die Parameter und übergibt diese an den Shortest Path Algorithmus der jeweiligen Datenbank. Nachdem die Route gefunden wurde, wird das Ergebnis dem Frontend übergeben.



Die beiden Datenbanken werden als Docker Container gemeinsam mittels Docker Compose gestartet. In den nächsten Abschnitten wird der Aufbau und die Realisierung der einzelnen Komponenten etwas näher erläutert.

Neo4J

Für Neo4J haben wir das **offizielle Docker Image** verwendet (https://hub.docker.com/_/neo4j). Bei der Installation und Konfiguration haben wir uns an die Dokumentation von neo4j gehalten. Die Daten, Logs & Plugins werden in einem eigenen Volume-Ordner persistiert, sodass diese nicht bei jedem Neustart des Containers verloren gehen.

Für die benötigen Graphalgorithmen muss das Neo4J Plugin **apoc** installiert werden. Dies muss im docker-compose.yml berücksichtigt werden. Zusätzlich ist uns aufgefallen, dass beim Hinzufügen der Knoten und Kanten recht viel Speicher im Heap benötigt wird. Mit den Default-Einstellungen ist bei uns die Datenbank paar mal abgestürzt, was durch eine weitere Einstellung verhindert werden kann. Die Konfigurationen im docker-compose.yml sahen final folgendermaßen aus:

```
environment:
    - NEO4J_AUTH=neo4j/geheim
    - NEO4J_dbms.memory.heap.max_size=6G
    - NEO4JLABS_PLUGINS=["apoc"]
```

Zum Finden des kürzesten Weges wurde die Funktion apoc.algo.aStar verwendet, welche den Weg mit dem A*-Algorithmus berechnet. Bei dieser muss man jedoch die tatsächlichen Start- und Zielknoten als Parameter übergeben (und nicht nur die Koordinaten). Das bedeutet, dass diese vorher mit zwei eigenen Queries gefunden werden müssen. Folgender Screenshot zeigt die Abfrage, bei der alle Nodes innerhalb einer bestimmten Entfernung zum gewünschten Punkt (z.B. 1km) durchsucht werden. Dieses Ergebnis wird anschließend aufsteigend nach Entfernung sortiert und auf einen Eintrag limitiert, sodass man nur den nächstgelegenen Node zu den gewünschten Koordinaten erhält.

```
CALL {{
    MATCH ({node_name}:Node)
    WITH {node_name},
        point.distance(
            point({{longitude:{node_name}.longitude, latitude:{node_name}.latitude}}),
            point({{longitude:{longitude}, latitude:{latitude}}})
        ) AS dist
    WHERE dist <= {search_distance}
    RETURN {node_name}
    ORDER BY dist ASC
    LIMIT 1
}}
```

Folgender Screenshot zeigt die Abfrage nach dem kürzesten Weg im Python Backend. Hier werden die Abfragen nach den nächstgelegenen Nodes zum Start- und Zielpunkt integriert. Bei der Funktion apoc.algo.aStar muss man die beiden Nodes, den Namen der Kante zwischen den Nodes (ROAD), das Kanten-Attribut für die Optimierung (length) und die Attribute für die Koordinaten der Nodes festlegen. Diese werden für die Berechnung der Heuristik im A*-Algorithmus benötigt.

```
query = f"""
    {self.find_nearest_node_query(source_longitude, source_latitude, "source")}
    WITH source
    {self.find_nearest_node_query(destination_longitude, destination_latitude, "destination")}
    WITH source, destination
    CALL apoc.algo.aStar (source, destination, 'ROAD', 'length', 'latitude', 'longitude')
    YIELD path, weight as costs
    RETURN path, costs, source, destination
"""
```

Tigergraph

Im Gegensatz zu Neo4J haben wir hier nicht das offizielle Image verwendet, da in den meisten Beiträgen das Image von xpertmind empfohlen wird (https://hub.docker.com/r/xpertmind/tigergraph). Bei diesem Image wurden paar Optimierungen hinsichtlich Speicherbedarf und Möglichkeiten beim Mounten durchgeführt.

Das Verwenden der Graphalgorithmen gestaltete sich hier deutlich schwieriger als bei Neo4J. Laut Dokumentation wird eine Funktion für den A*-Algorithmus zur Verfügung gestellt (https://docs.tigergraph.com/graph-ml/current/pathfinding-algorithms/a-star). Diese muss jedoch unter Verwendung des Docker Containers zuvor heruntergeladen und anschließend über ein Volume in das Image geladen werden. Anschließend muss man diese über einen GSQL-Befehl installieren. Dies funktionierte jedoch zu Beginn nicht, da diese Funktion einige Syntax-Fehler hatte, die wir ausbessern mussten.

Ähnlich wie bei Neo4J mussten wir eine eigene Query schreiben, welche den nächstgelegenen Node zu gegebenen Koordinaten findet. Diese sieht folgendermaßen aus:

```
BEGIN
    USE GRAPH myGraph
    CREATE OR REPLACE QUERY find_nearest_node (DOUBLE lon, DOUBLE lat, DOUBLE search_distance) FOR GRAPH myGraph SYNTAX v2 {
        nodes = {Node.*};
        filtered_nodes =
            SELECT n FROM nodes:n
            WHERE tg_GetDistance(n.lat,n.lon,lat,lon) < search_distance
            ORDER by tg_GetDistance(n.lat,n.lon,lat,lon) ASC
            LIMIT 1;
        PRINT filtered_nodes;
    }
END
INSTALL QUERY find_nearest_node
```

Nachdem der richtige Start- und Zielknoten gefunden wurde, kann man die IDs dieser Knoten an die A*-Funktion mitgeben. Wie in folgendem Screenshot ersichtlich muss man hier wie bei Neo4J die beiden Knoten, den Namen der Beziehung zwischen den Knoten, den Datentyp des Kantengewichts, die Attribute für die Koordinaten eines Nodes und das Kantenattribut für das Gewicht mitgeben.

```
result = self.execute_tigergraph_query(f'''
    BEGIN
        RUN QUERY tg_astar(({src_node_id},"Node"), ({dest_node_id},"Node"), ["Road"], "DOUBLE", "lat" , "lon", "length")
    END
''')
```

Route Service Backend

Die Aufgabe des Python Route Service Backends ist es, Anfragen entgegenzunehmen und diese an die jeweilige Datenbanken weiterzuleiten. Für die Kommunikation mit dem Backend haben wir eine REST-Schnittstelle mit Flask entwickelt. Diese unterstützt die folgenden Routen:

GET /route → kürzesten Weg zwischen zwei Punkten finden

- Parameter → src_lon, src_lat, dest_lon, dest_lat & db_service (neo4j oder tigergraph)
- Result
    - Status Code 200 → JSON bestehend aus
        - path → Liste der Nodes auf dem Weg mit Koordinaten
        - path_costs → Länge des Weges in Meter
        - source_node → Verwendeter Startpunkt
        - destination_node → Verwendeter Zielpunkt
        - duration → Dauer der Suchvorganges in Sekunden
    - Status Code 400 → Fehlermeldung, weil keine Route gefunden wurde

GET /reset/neo4j → Auffüllen / Zurücksetzen der Daten in Neo4J

GET /reset/tigergraph → Auffüllen / Zurücksetzen der Daten in TigerGraph

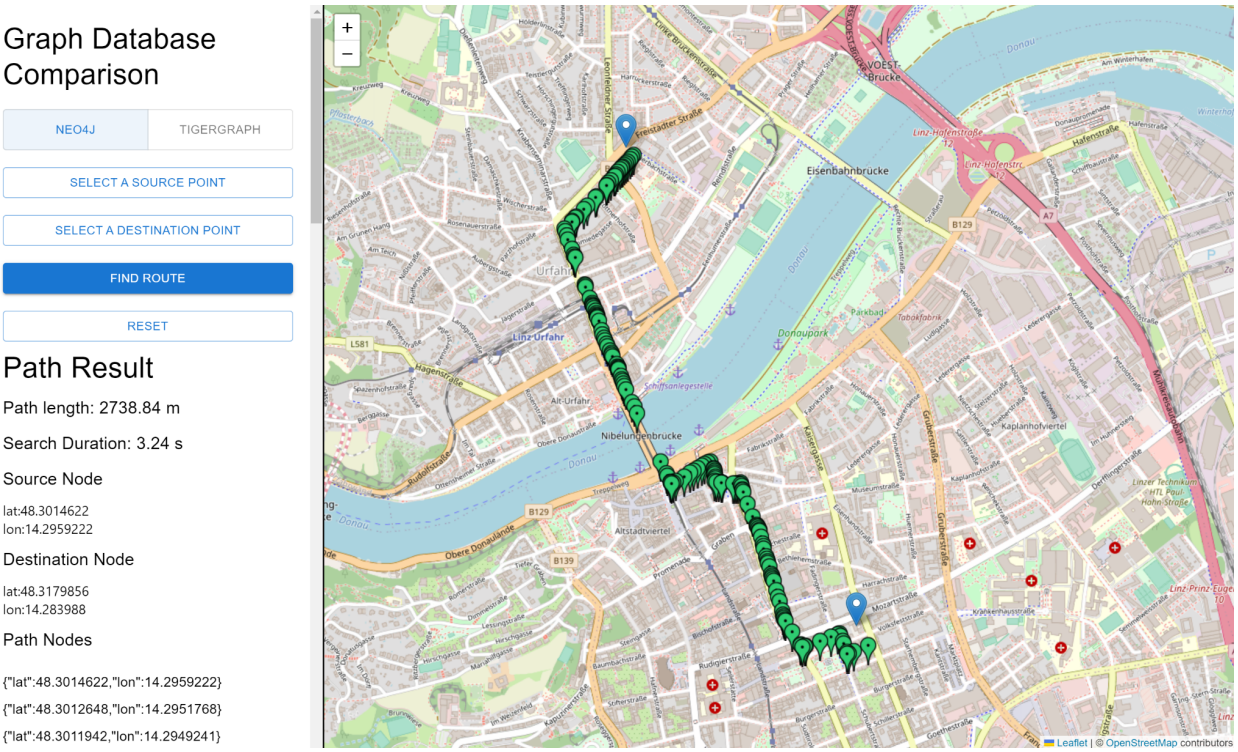
Das Skript wird durch den Befehl python server.py im Ordner route_service gestartet. Der REST Server wird anschließend auf Port 5000 gestartet und wartet auf Requests. Für die Interaktionen mit den beiden Datenbanken haben wir eigene Clients geschrieben, welche das Auffüllen der Datenbank und das Finden des kürzesten Weges unterstützen.

Route Service Frontend

Damit die Funktionalität des Backends etwas intuitiver verwendet werden kann, haben wir uns entschieden, ein Frontend mit React zu entwickeln. Durch Ausführen von **npm start** wird die Webseite auf **localhost:4000** gestartet.

In folgender Abbildung befindet sich ein Screenshot der Webseite. Im linken Bereich befindet sich eine Toolbar und auf der rechten Seite befindet sich die Karte. Ganz oben in der Toolbar gibt es einen Switch, bei dem man zwischen den beiden Datenbanken hin- und herschalten kann. Darunter befindet sich sich die Buttons **Select a Source Point & Select a Destination Point**. Mit einem Klick auf diese kann man den jeweiligen Punkt auf der Karte festlegen. Der ausgewählte Punkt wird auf der Karte mit einem blauen Marker visualisiert. Wenn man beide Punkte festgelegt hat, kann man durch einen Click auf den Button **Find Route** den kürzesten Weg zwischen diesen Punkten berechnen lassen. Wenn ein Ergebnis gefunden wird, wird der Weg auf der Karte mit grünen Markern visualisert und in der Toolbar werden unterschiedliche Informationen angezeigt:

- Path length → Länge des Weges in Meter
- Search duration → Dauer der Datenbankabfrage in Sekunden
- Source Node → Koordinaten des Startpunktes (welcher für die Suche verwendet wurde)
- Destination Node → Koordinaten des Zielpunktes (welcher für die Suche verwendet wurde)
- Path Nodes → Liste mit allen Nodes (Koordinaten) auf dem Weg (entsprechen den grünen Markern auf der Karte)



Fazit

Wir haben in diesem Projekt versucht, die Verwendbarkeiten der graphbasierten Datenbanken Neo4J und TigerGraph zu untersuchen und diese miteinander zu vergleichen. Im Großen und Ganzen können wir behaupten, dass für unseren Use Case die Arbeit mit Neo4J deutlich leichter und unkomplizierter war. Bei TigerGraph wird zwar ein A*-Algorithmus zur Verfügung gestellt, dieser hatte jedoch Syntax-Fehler und ist nicht gut implementiert, da er unglaublich lange für die Berechnung des kürzesten Weges braucht (ca. 5 mal so lange wie Neo4J). Wir bezweifeln ebenfalls, dass der Algorithmus von TigerGraph richtige Ergebnisse liefert, da diese stark von den Ergebnissen von Neo4J abweichen. Dies kann eventuell an der Floating Point Precision liegen. Aus irgendeinem Grund werden die Koordinaten der Nodes bei TigerGraph mit 4 Nachkommastellen angegeben, bei Neo4J sind es 8.

Bei Neo4J konnte man die Library für die Graphalgorithmen ganz einfach mit einem Eintrag im docker-compose.yml installieren. Bei TigerGraph mussten wir das Skript von GitHub herunterladen, ausbessern, das File mit einem Volume in den Container laden und anschließend noch über eine Query diese Funktion in die Datenbank laden.