

Informe

Fuentes de datos

Scheduled

<https://pid.cz/en/opendata/#h-gtfs>

En la siguiente carpeta de drive y en el siguiente repositorio de Github se almacenarán todos los archivos necesarios para realizar todo lo enunciado en el informe

La descarga fue realizada el 28 de junio.

De los datos descargados de la página, únicamente nos quedaremos con los siguientes archivos .txt . Estos se pueden encontrar en el archivo PID_GTFS.tar.gz.

- agency.txt
- calendar_dates.txt
- calendar.txt
- feed_info.txt
- levels.txt
- pathways.txt
- routes.txt
- shapes.txt
- stop_times.txt
- stops.txt
- transfers.txt
- trips.txt

Real Time

Obtenidos de golemio (<https://api.golemio.cz/pid/docs/openapi/index.htm#/>)

Dentro del sitio se ofrecen varios endpoints para obtener información realtime.

Por un lado, hay una API de protobuf en /v2/vehiclepositions/gtfsrt/vehicle_positions.pb .

En un principio se intentó utilizar los datos de ese endpoint, pero la información era de muy mala calidad: llegaban puntos de pocos vehículos a intervalos muy irregulares, lo que dificultó mucho armar las trayectorias

También el sitio ofrece una API REST en /v2/public/vehiclepositions .

Esta es la que se terminó utilizando, debido a que fue mucho mas consistente en la calidad y frecuencia de los datos.

GTFS Schedule

Importación de datos

Se asume que ya se tiene una base Postgres con el plugin de MobilityDB instalado corriendo en el puerto 25432, con usuario postgres, sin contraseña, y con una base de datos prague armada.

Antes de comenzar a realizar las inserciones en la base de datos, se debe ejecutar el siguiente comando

```
ALTER DATABASE prague SET timezone TO 'UTC'
```

con el objetivo de unificar el manejo de fechas y horas. Esto garantiza que todos los timestamps sean interpretados en horario UTC, evitando inconsistencias derivadas de zonas horarias locales y asegurando una correcta comparación y análisis temporal de los datos de transporte.

Se debe instalar el paquete de gtfs-via-postgres para importar la información de GTFS a postgres.

```
npm install -g gtfs-via-postgres
```

Se definen las variables de entorno listadas a continuación para que las herramientas de CLI de PostgreSQL puedan conectarse correctamente a la base de datos. Se deben cambiar en caso de ser necesario (por ejemplo el PGPORT si no se usa docker).

```
export PGUSER=postgres;
export PGHOST=localhost;
```

```
export PGPORT=25432;
export PGDATABASE=prague;
```

Luego de haber descomprimido el [PID_GTFS.tar.gz](#) sobre algún directorio, se corre allí el siguiente comando para cargar toda la información a la base de datos.

```
npm exec -- gtfs-to-sql --require-dependencies -- *.txt | psql -b
```

Limpieza

En la documentación (<https://gtfs.org/documentation/schedule/reference>), se observa que los campos geográficos en las tablas GTFS vienen especificados en formato latitud-longitud.

<code>stop_lat</code>	Latitude	Conditionally Required	Latitude of the location. For stops/platforms (<code>location_type=0</code>) and boarding area (<code>location_type=4</code>), the coordinates must be the ones of the bus pole — if exists — and otherwise of where the travelers are boarding the vehicle (on the sidewalk or the platform, and not on the roadway or the track where the vehicle stops). Conditionally Required: - Required for locations which are stops (<code>location_type=0</code>), stations (<code>location_type=1</code>) or entrances/exits (<code>location_type=2</code>). - Optional for locations which are generic nodes (<code>location_type=3</code>) or boarding areas (<code>location_type=4</code>).
<code>stop_lon</code>	Longitude	Conditionally Required	Longitude of the location. For stops/platforms (<code>location_type=0</code>) and boarding area (<code>location_type=4</code>), the coordinates must be the ones of the bus pole — if exists — and otherwise of where the travelers are boarding the vehicle (on the sidewalk or the platform, and not on the roadway or the track where the vehicle stops). Conditionally Required: - Required for locations which are stops (<code>location_type=0</code>), stations (<code>location_type=1</code>) or entrances/exits (<code>location_type=2</code>). - Optional for locations which are generic nodes (<code>location_type=3</code>) or boarding areas (<code>location_type=4</code>).

Atributos de donde se obtienen las coordenadas de cada stop según GTFS

<code>shape_pt_lat</code>	Latitude	Required	Latitude of a shape point. Each record in shapes.txt represents a shape point used to define the shape.
<code>shape_pt_lon</code>	Longitude	Required	Longitude of a shape point.

Atributos de donde se obtienen las coordenadas de cada shape según GTFS

La herramienta gtfs-via-postgres automáticamente castea estos campos al tipo `geography` de PostgreSQL al momento de la importación, bajo el sistema de referencia EPSG:4326 (WGS 84).

Además crea una view `shapes_aggregates`

Por comodidad se convierten estos datos al sistema EPSG:5514 después de la importación.

Limpieza inicial y determinación de fechas

Se analizaron los SRIDs y que no haya geometrías en NULL.

```
CREATE EXTENSION mobilitydb CASCADE; -- Asi instala tambien postgis
```

```

SELECT DISTINCT ST_SRID(stop_loc) FROM stops;
SELECT DISTINCT ST_SRID(shape_pt_loc) FROM shapes;
-- Ambos retornan 4326

SELECT COUNT(*) FROM stops WHERE stop_loc IS NULL;
SELECT COUNT(*) FROM shapes WHERE shape_pt_loc IS NULL;
-- Se verifica que no hay geometrias vacías (ambos dan 0)

```

Luego, se arma la view `service_dates`: a partir de la información de `calendar` (que contiene los días en los que un servicio está disponible), y `calendar_dates`, que aporta información sobre feriados y otras excepciones, se computa día por día todas las fechas en las que los servicios están disponibles.

```

-- Se arma una view con las fechas donde todos los servicios estan disponibles
-- Basado en https://github.com/pabloito/MDB-Importer
DROP VIEW IF EXISTS service_dates;
CREATE VIEW service_dates AS (
    SELECT service_id, date_trunc('day', d)::date AS date
    FROM calendar c, generate_series(start_date, end_date, '1 day'::interval) AS d
    WHERE (
        (monday = 'available' AND extract(isodow FROM d) = 1) OR
        (tuesday = 'available' AND extract(isodow FROM d) = 2) OR
        (wednesday = 'available' AND extract(isodow FROM d) = 3) OR
        (thursday = 'available' AND extract(isodow FROM d) = 4) OR
        (friday = 'available' AND extract(isodow FROM d) = 5) OR
        (saturday = 'available' AND extract(isodow FROM d) = 6) OR
        (sunday = 'available' AND extract(isodow FROM d) = 7)
    )
    EXCEPT
    SELECT service_id, date
    FROM calendar_dates WHERE exception_type = 'removed'
    UNION
    SELECT c.service_id, date
    FROM calendar c JOIN calendar_dates d ON c.service_id = d.service_id
    WHERE exception_type = 'added' AND start_date <= date AND date <= end_date
);

```

A continuación, se convierten a geometry los stops y se arma la tabla trajectories, que contiene todos los shapes de cada trip, ambas cosas en SRID 5514.

```

ALTER TABLE stops
ALTER COLUMN stop_loc TYPE geometry(Point, 5514)
USING ST_Transform(stop_loc::geometry, 5514);

CREATE TABLE trajectories (
    shape_id text,
    traj geometry(LineString, 5514)
);

INSERT INTO trajectories (shape_id, traj)
SELECT shape_id, ST_Transform(ST_SetSRID(shape, 4326), 5514)
FROM shapes_aggregated;

```

Luego, se ejecuta el script `mdb_importer_scheduled.sql` para poder trabajar con los datos de Praga a partir de las tablas generadas por `gtfs-via-postgres`. Este fue adaptado y actualizado del script de Godfrid y Radnic, ya que la estructura de las tablas generadas por `gtfs-via-postgres` no es igual a la que proponen ellos, además de que la sintaxis de MobilityDB cambió desde entonces.

Este script, generan los segmentos en base a los stops, y finalmente se arman los trips para cada fecha que se indique en `service_dates`.

Funciona a través de las siguientes tablas:

1. `trips_stops` : contiene cada uno de los stops, indicando el orden dentro del trip, su stop_id. y el tiempo de llegada. Se calcula el porcentaje recorrido sobre la trayectoria total para luego poder proyectar sobre el shape.
2. `trip_segs` : almacena información sobre los segmentos entre pares de paradas consecutivas de un mismo viaje, incluyendo los porcentajes que indican la posición sobre la trayectoria. Para cada parada, se usa una ventana para crear un segmento hacia la siguiente, asociando los datos relevantes y calculando el subtramo exacto en la geometría. Además, registra la longitud y el número de puntos de cada segmento, eliminando los casos donde no se pudo calcular la geometría.
3. `trip_points` : contiene puntos intermedios entre paradas de un trip, generados a partir de la geometría de los segmentos. Para cada punto, se calcula su posición relativa a lo largo del segmento y asigna una hora de llegada interpolada entre las horas de las paradas inicial y final.
4. `trips_input` : almacena los puntos de cada trip armando el timestamp en el que pasa por cada punto. Restringe los registros a una fecha específica para no hacer que la generación de trips sea demasiado costosa.
5. `trips_mdb` : a partir de los puntos y timestamps de `trips_input`, arma los tgeompointseqs que representan el trip definitivo. Se toma solo el primer punto de cada `trip_id`, `route_id` y `date` de cada timestamp, ya que hay datos repetidos.

Análisis de los tipos de rutas

Siguiendo la [referencia](#), se hizo un análisis de los tipos de rutas presentes en la base de datos

```
WITH route_types(route_type, name) AS (
    SELECT '0', 'streetcar' UNION
    SELECT '1', 'subway' UNION
    SELECT '2', 'rail' UNION
    SELECT '3', 'bus' UNION
    SELECT '4', 'ferry' UNION
    SELECT '5', 'cable tram' UNION
    SELECT '6', 'aerial' UNION
    SELECT '7', 'funicular' UNION
    SELECT '11', 'trolley' UNION
    SELECT '12', 'monorail'
),
route_groups AS
(
    SELECT
        route_type,
        COUNT(*) AS qty,
        ROUND(COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (), 2) AS perc
    FROM routes
    GROUP BY route_type
)
SELECT name, qty, perc
FROM route_groups g JOIN route_types t ON g.route_type = t.route_type
ORDER BY perc DESC
```

	name text	qty bigint	perc numeric
1	bus	670	83.23
2	rail	88	10.93
3	streetcar	36	4.47
4	ferry	6	0.75
5	subway	3	0.37
6	trolley	2	0.25

Cantidades y porcentajes de cada route_type

Notar que la suma de los porcentajes da 100%.

A continuación se mostrarán las rutas filtradas por tipo, restringiendo la región únicamente a un polígono que representa la ciudad de Praga.

El shapefile se obtuvo de osmtoday en el siguiente [link](#). Se provee el archivo [praha.geojson](#) en la carpeta de drive.

Para importarlo se puede utilizar la herramienta ogr2ogr (cambiando los parámetros en caso de ser necesario).

```
ogr2ogr -f "PostgreSQL" \
PG:"host=localhost dbname=prague port=25432 user=postgres" \
praha.geojson \
-nln province \
-nlt MULTIPOLYGON \
-lco GEOMETRY_NAME=geom \
-lco FID=id \
-overwrite
```

Debido a que los caminos pueden entrar, salir o ser tangenciales al polígono, si se quiere obtener las trayectorias que caigan dentro de él, se debe tener en cuenta que se pueden obtener [MultiLineStrings](#) o incluso [GeometryCollections](#).

Se muestra la cantidad de cada uno, el 8 de julio, al realizar la operación de intersección.

```
SELECT ST_GeometryType(geom_part) AS geom_type, COUNT(*) AS qty
FROM (
  SELECT ST_Intersection(
    ST_Transform(geom, 5514),
    traj)
  AS geom_part
  FROM trips_mdb, province
  WHERE ST_Intersects(traj, ST_Transform(geom, 5514))
) AS parts
GROUP BY ST_GeometryType(geom_part);
```

	geom_type text	qty bigint
1	ST_GeometryCollection	16
2	ST_LineString	22623
3	ST_MultiLineString	8516

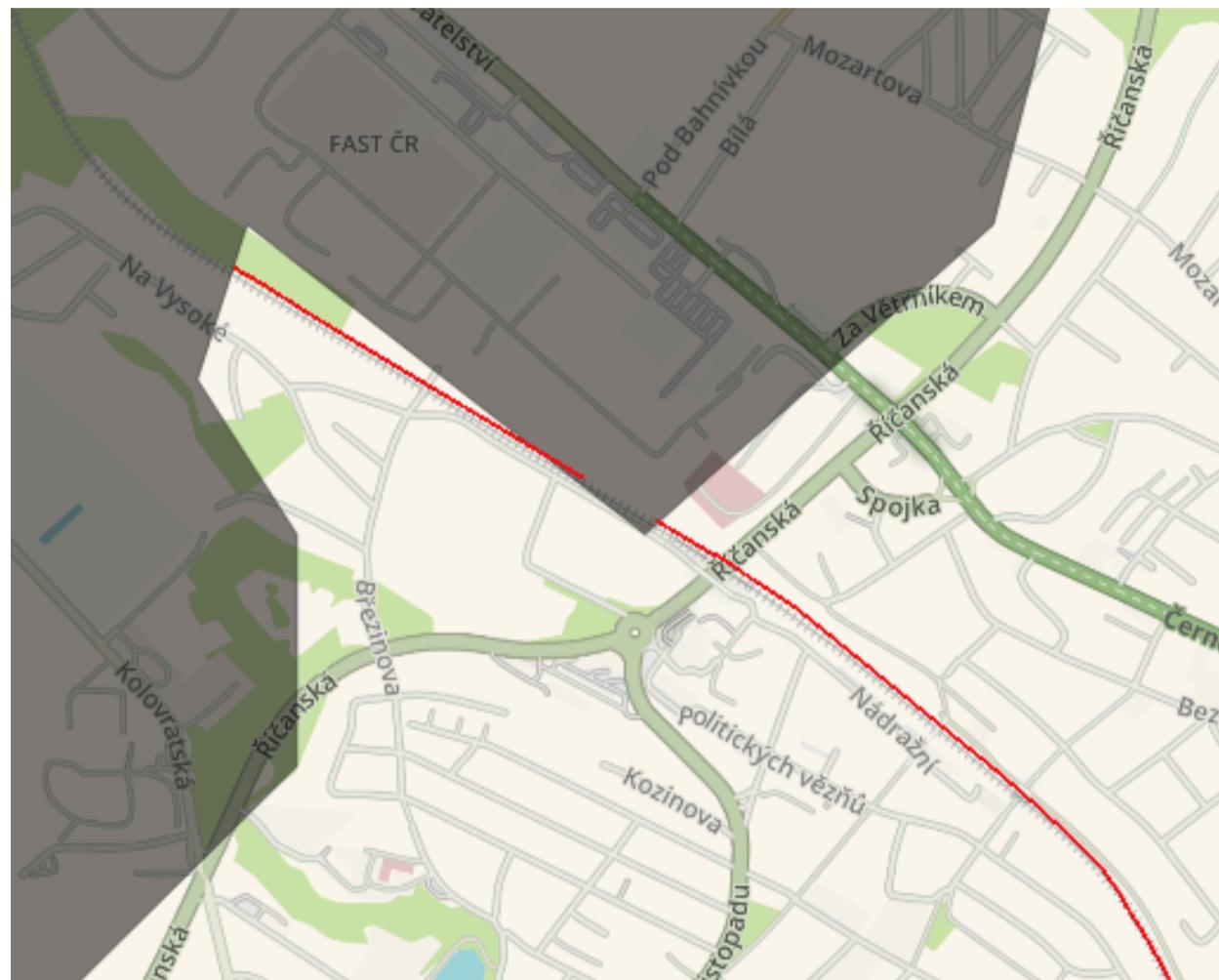
Se puede observar que la mayoría de las trayectorias no entran y salen del polígono, aunque hay una buena parte que si lo hace (en su mayor parte no de manera tangencial).

```
DROP MATERIALIZED VIEW IF EXISTS MultilineShapeTrips;
CREATE MATERIALIZED VIEW MultilineShapeTrips AS
SELECT ROW_NUMBER() OVER () as id, trip_id, ST_GeometryType(ST_Intersection(traj, ST_Transform(geom, 5514))), S
T_Difference(
```

```

traj,
ST_Transform(geom, 5514)
)
AS geom_part
FROM trips_mdb, province
WHERE ST_Intersects(traj, ST_Transform(geom, 5514)) AND ST_GeometryType(ST_Intersection(traj, ST_Transform(ge
om, 5514))) = 'ST_MultiLineString'

```



Visualización de trayectoria que entra y luego sale del polígono de Praga

Ahora, se creará una view para las trayectorias de las rutas de los distintos tipos de transportes, procurando ignorar todas las geometrías que no son líneas (se vio que solo 37 trayectorias tienen puntos).

```

DROP TABLE IF EXISTS prague_trajectories CASCADE;
CREATE TABLE prague_trajectories (
    id text,
    trip_id text,
    traj geometry(MultiLineString, 5514),
    route_id text,
    route_type route_type_val
);

INSERT INTO prague_trajectories
SELECT trip_id || date as id, trip_id, ST_CollectionExtract(ST_Intersection(traj, ST_Transform(geom, 5514)), 2) as traj, r.r
oute_id, r.route_type
FROM trips_mdb t, province, routes r
WHERE ST_Intersects(traj, ST_Transform(geom, 5514)) AND r.route_id=t.route_id;

DROP VIEW IF EXISTS bus_trajectories;
CREATE VIEW bus_trajectories AS
SELECT id, trip_id, t.traj FROM prague_trajectories t
WHERE route_type='3';

DROP VIEW IF EXISTS rail_trajectories;
CREATE VIEW rail_trajectories AS
SELECT id, trip_id, t.traj FROM prague_trajectories t

```

```

WHERE route_type='2';

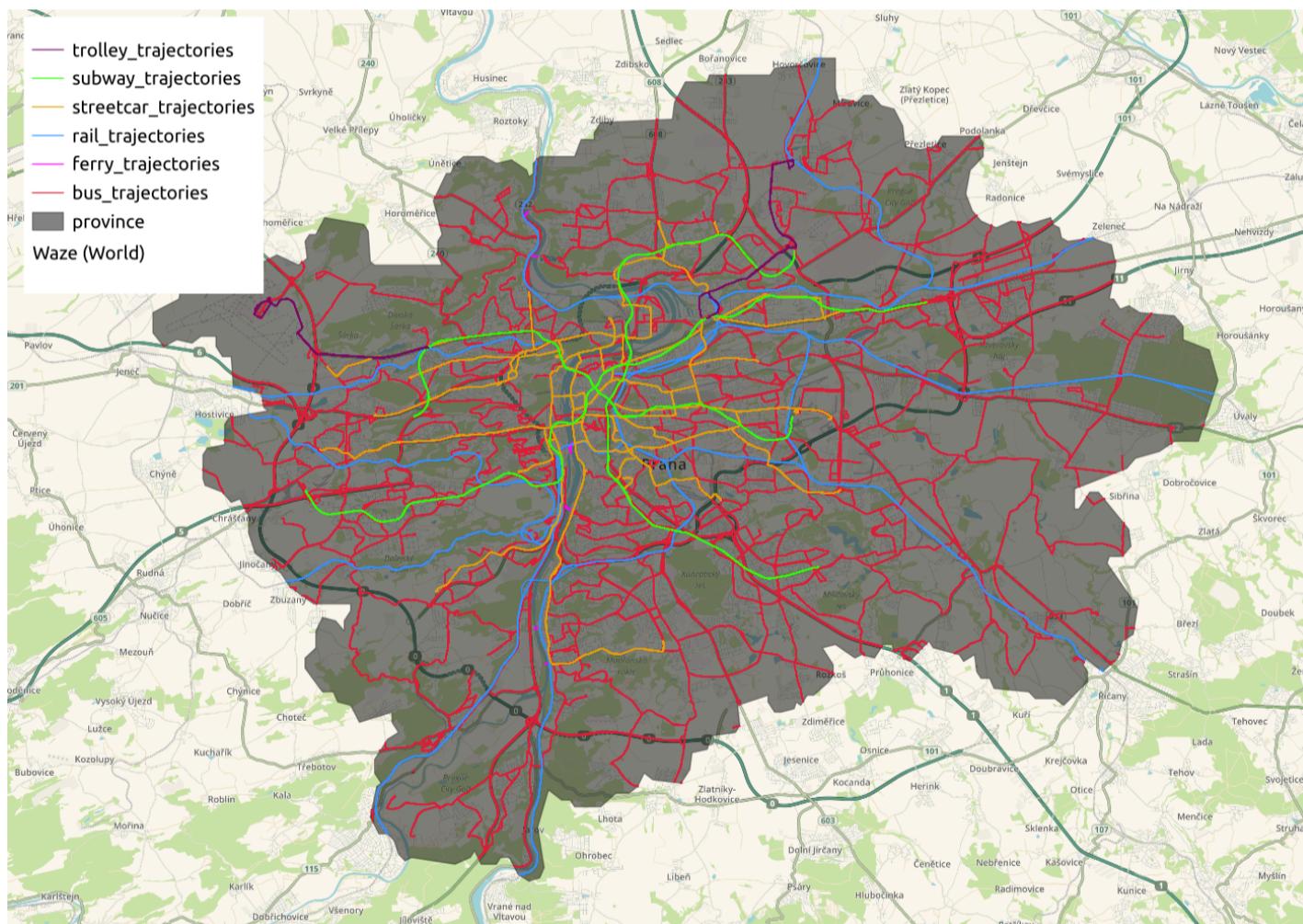
DROP VIEW IF EXISTS streetcar_trajectories;
CREATE VIEW streetcar_trajectories AS
SELECT id, trip_id, t.traj FROM prague_trajectories t
WHERE route_type='0';

DROP VIEW IF EXISTS ferry_trajectories;
CREATE VIEW ferry_trajectories AS
SELECT id, trip_id, t.traj FROM prague_trajectories t
WHERE route_type='4';

DROP VIEW IF EXISTS subway_trajectories;
CREATE VIEW subway_trajectories AS
SELECT id, trip_id, t.traj FROM prague_trajectories t
WHERE route_type='1';

DROP VIEW IF EXISTS trolley_trajectories;
CREATE VIEW trolley_trajectories AS
SELECT id, trip_id, t.traj FROM prague_trajectories t
WHERE route_type='11';

```



Visualización de trayectorias de troleys, subways, streetcars, rails, ferries y buses

Agregación de rutas por segmento

La consulta sigue el ejemplo de la página 486 del libro de la cátedra. Se realiza un filtrado por `route_type = '3'` para incluir solo colectivos y se limita a los segmentos que comenzaron el 8 de julio de 2025 entre las 15:00 y las 17:00.

El join con la tabla `trips` es necesario porque `trip_segs` no tiene el campo `direction_id`: si lo tuviera, cada parada aparecería duplicada para dirección 0 y 1, ya que `trip_segs` se genera a partir de `trip_stops`.

```

DROP MATERIALIZED VIEW IF EXISTS SegmentsDisplay;
CREATE MATERIALIZED VIEW SegmentsDisplay AS
SELECT
    c.from_stop_id || c.to_stop_id as id,

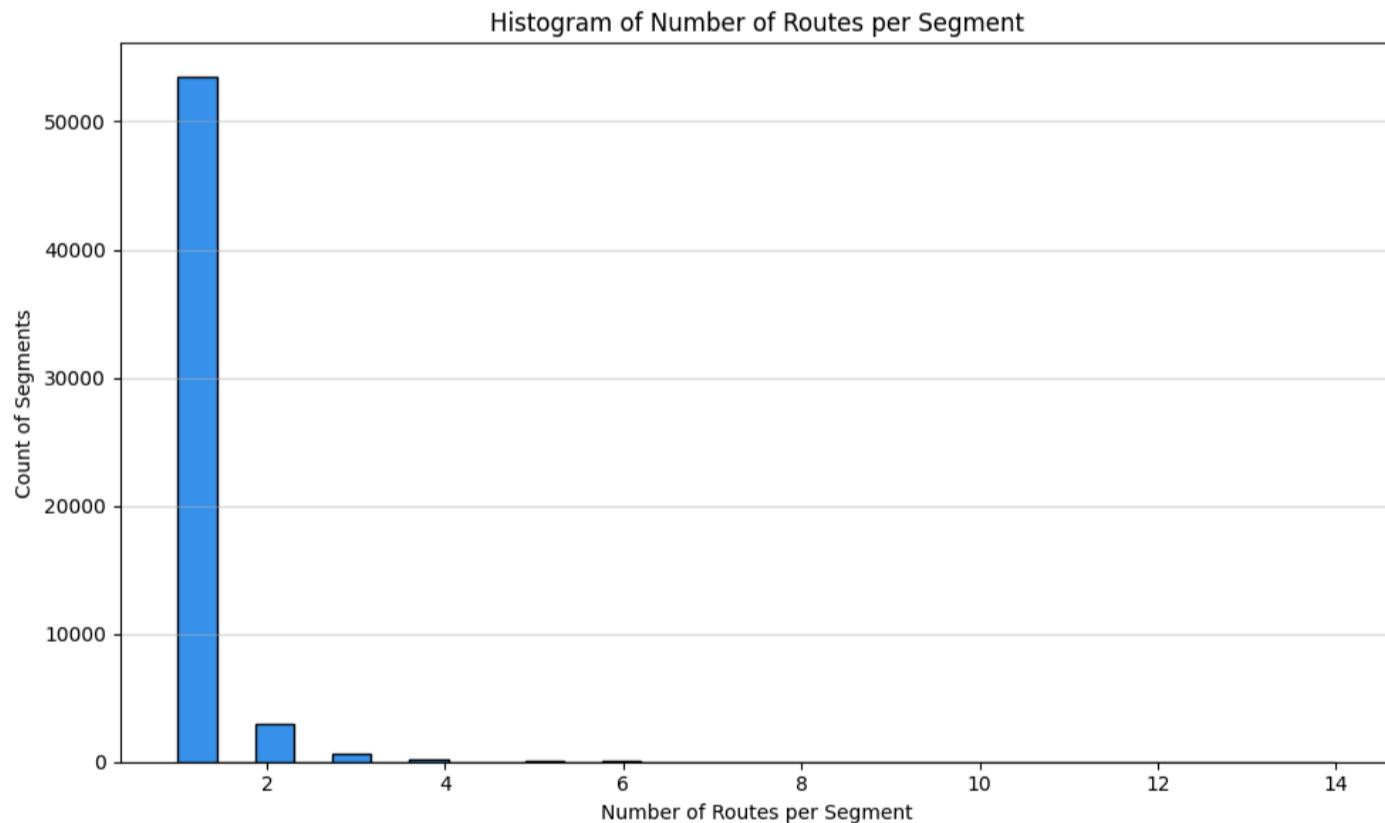
```

```

s.seg_geom,
COUNT(DISTINCT c.route_id) AS num_routes
FROM
trip_segs s,
connections c,
trips t
WHERE
t.trip_id = s.trip_id
AND s.route_id = c.route_id
AND t.direction_id = c.direction_id
AND s.stop1_sequence = c.from_stop_sequence
AND s.stop2_sequence = c.to_stop_sequence
AND date = '2025-07-08'
AND route_type = '3'
AND stop1_arrival_time BETWEEN '15:00:00' AND '17:00:00'
GROUP BY
c.from_stop_id,
c.from_stop_name,
c.to_stop_id,
c.to_stop_name,
s.seg_geom;

```

Se puede visualizar un histograma de la frecuencia de numero de rutas por segmento ejecutando el script [agg_routes_per_segment.py](#).



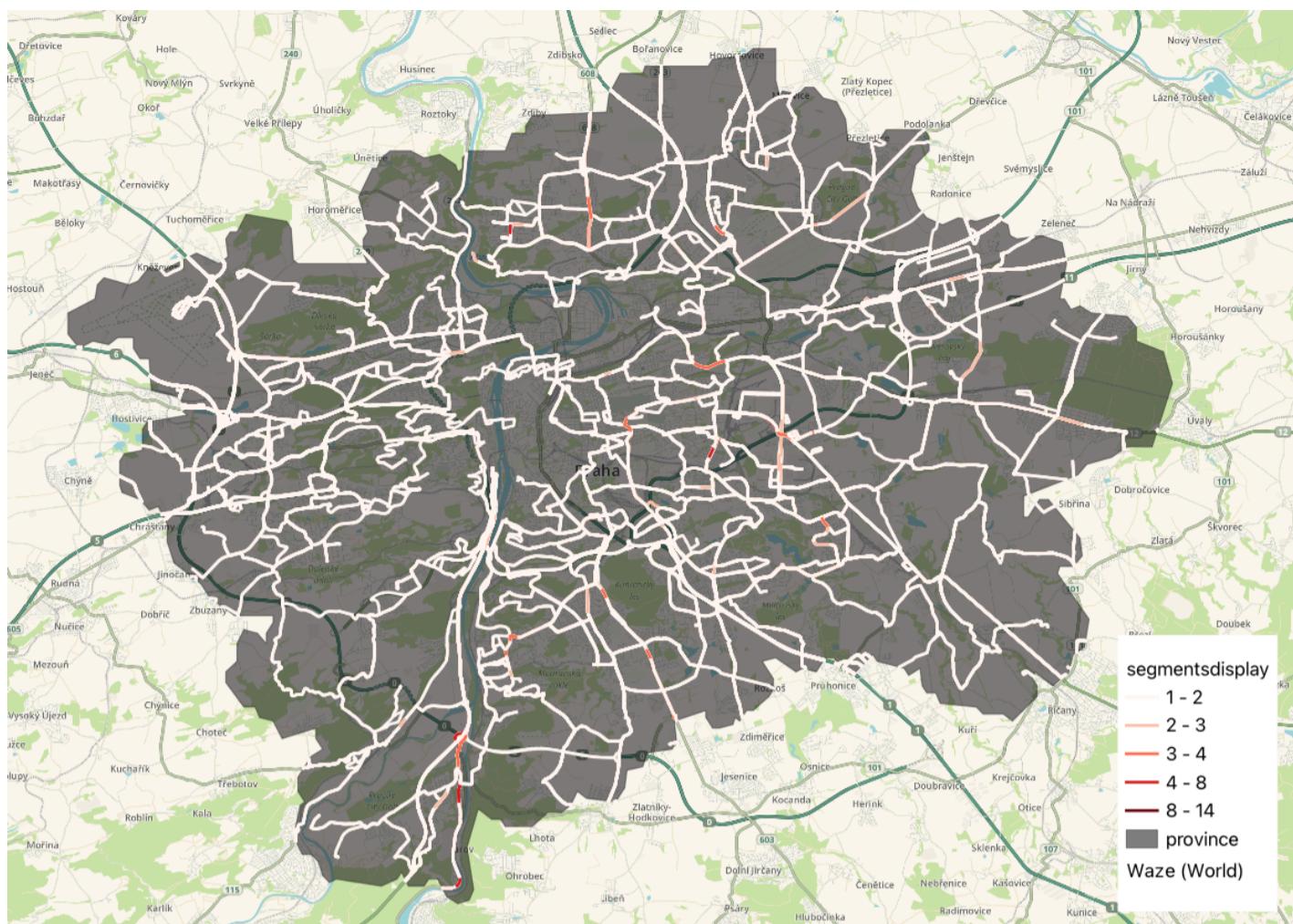
Histograma que muestra la cantidad de rutas por segmento

El histograma presentado muestra la distribución de la cantidad de rutas que utilizan cada segmento de la red de transporte analizada. En el eje horizontal se observa el número de rutas que pasan por un mismo segmento, mientras que en el eje vertical se representa la cantidad de segmentos que presentan esa característica.

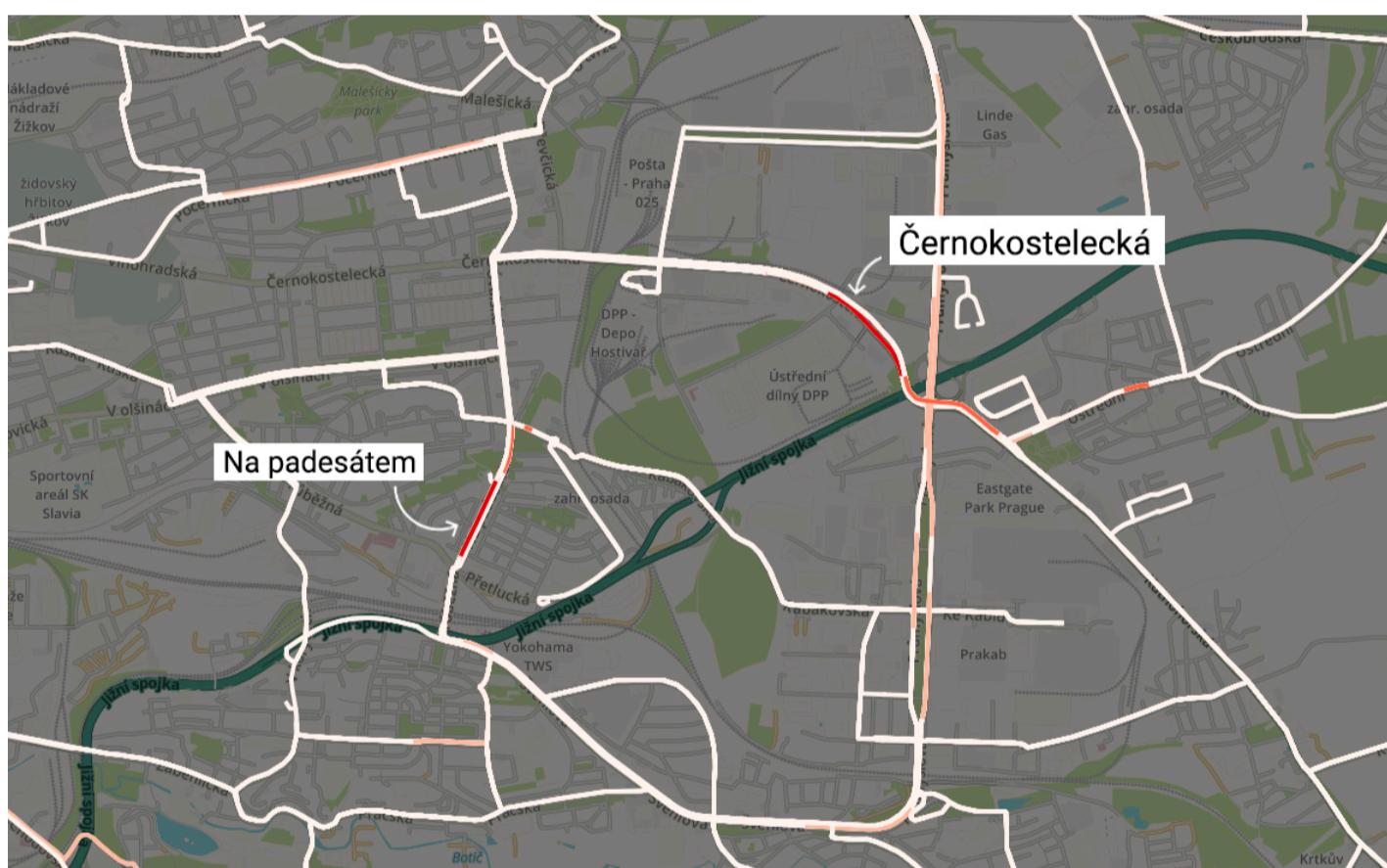
El gráfico evidencia que la gran mayoría de los segmentos son exclusivos de una sola ruta. A medida que aumenta la cantidad de rutas que comparten un segmento, la frecuencia disminuye de forma abrupta, reflejando que los segmentos compartidos por dos o más rutas son una minoría dentro de la red.

Esta distribución es esperable en sistemas de transporte donde la mayor parte de la infraestructura está dedicada a recorridos específicos, y solo algunos corredores principales funcionan como nexos para varias líneas o rutas.

```
SELECT id, seg_geom, num_routes FROM SegmentsDisplay
```



Visualización de cantidad de rutas por segmento en el mapa de Praga. Mientras más claro es el segmento, menor cantidad de rutas tendrá.



Identificación de las rutas Na padesátem y Černokostelecká, que cuentan con una alta cantidad de rutas

Para evaluar el nivel de tránsito en la ciudad, se identificaron las calles que aparecen en rojo en el mapa, ya que este color señala las áreas de mayor densidad vehicular. En este análisis, se destacaron especialmente Černokostelecká y Na padesátem.

Černokostelecká es una vía principal de la ciudad, con varios carriles y control de velocidad. Es utilizada tanto para el tránsito local como para el cruce de la ciudad, lo que genera un flujo vehicular alto o moderado a lo largo del día.

Na padesátem, por su parte, actúa como un nodo de intercambio entre diversas líneas de bus, tranvía, tren y metro. Esto favorece un flujo constante de tránsito, especialmente en las horas pico, debido al gran volumen de personas que utilizan el transporte público.

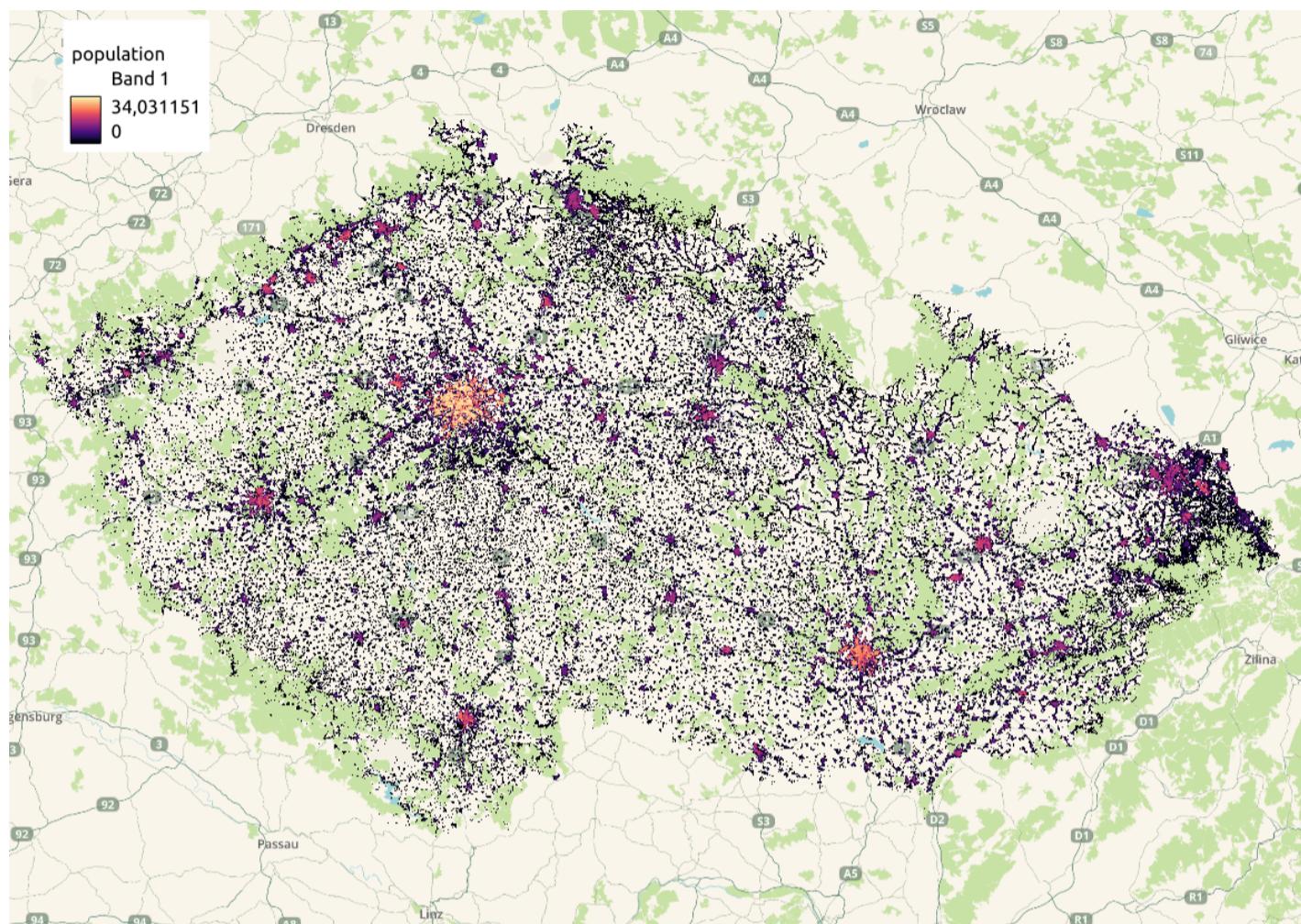
Análisis de los trips por la ciudad en relación con la población

Se debe instalar la extensión `postgis_raster` de la siguiente manera

```
CREATE EXTENSION postgis_raster
```

Se necesitará un raster con los datos de población de Praga. Este se podrá encontrar [esta carpeta de drive](#), bajo el nombre `population.tif`.

Se obtuvo de la siguiente fuente: <https://hub.worldpop.org/geodata/summary?id=54434>



Raster de población en toda República Checa

Observar como destaca Praga por ser la capital y por lo tanto la zona con mayor densidad de población. Aunque también se puede ver como por ejemplo Brno, Ostrava y Pilsen la siguen en la lista de ciudades con más población (los puntos rosas/violetas que destacan)

Luego, se deben correr en la terminal los dos siguientes comandos, sobre el directorio donde se encuentra el archivo:

```
raster2pgsql -s 4326 -I -C population.tif > population.sql
```

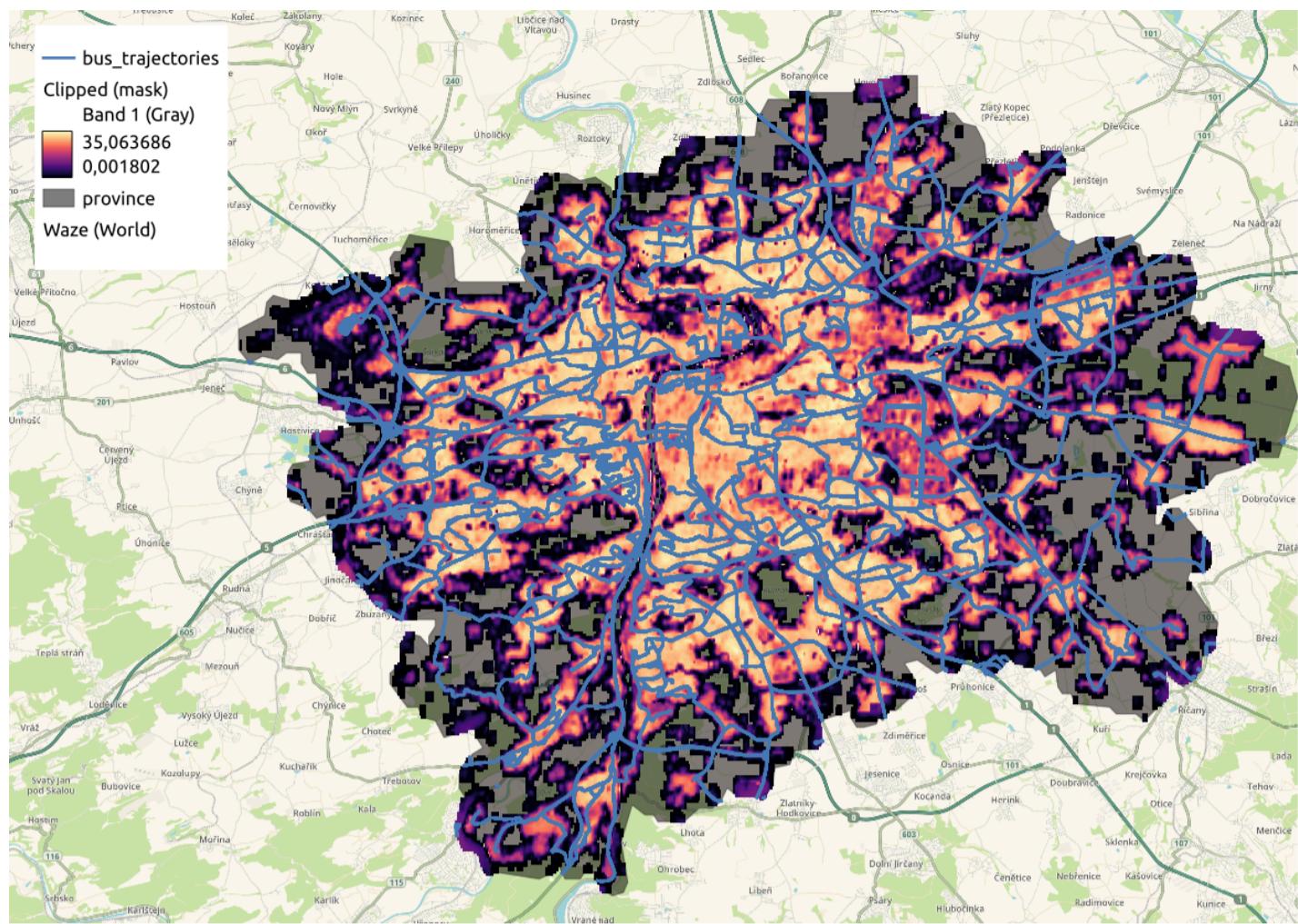
```
psql prague -U postgres -p 25432 -h localhost < population.sql
```

Ahora, se debe clippear el raster de la población con el de la ciudad de Praga, de la siguiente manera:

```
CREATE TABLE prague_pop AS
  WITH clipgeo AS (
    SELECT ST_Transform(geom, 4326) AS provincegeo
      FROM province LIMIT 1
  )
  SELECT ST_Clip(a.rast::raster, 1, c.provincegeo, true) AS st_clip
    FROM population a, clipgeo c
  );
```

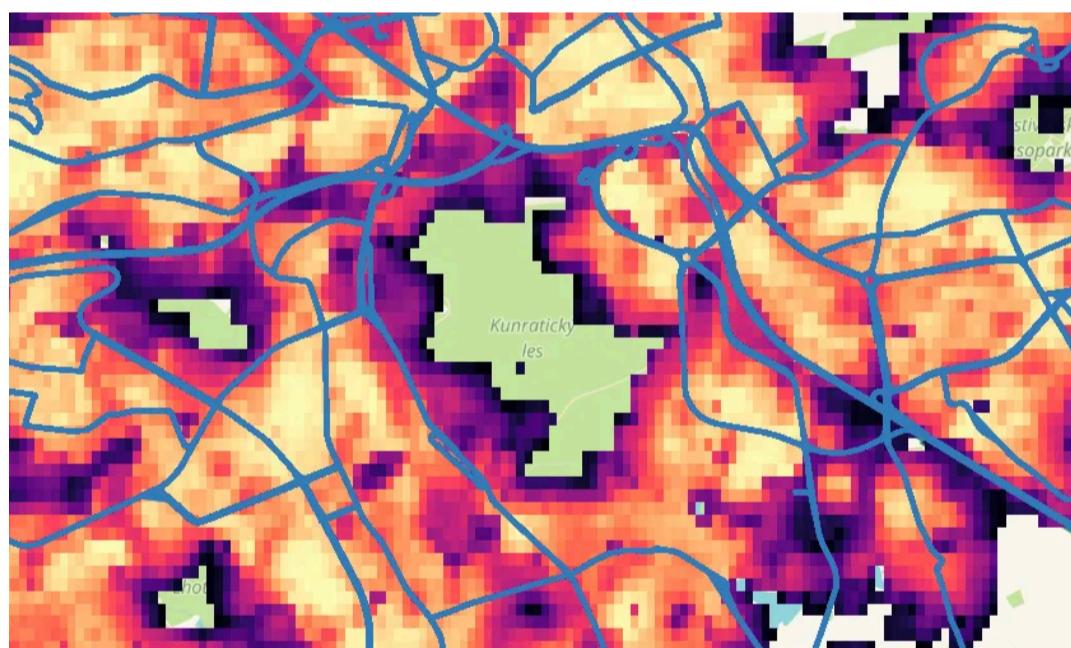
```
ALTER TABLE prague_pop RENAME COLUMN st_clip TO rast;
```

Si graficamos el raster al mismo tiempo que las trayectorias de colectivos obtenidas previamente (la view `bus_trajectories`) queda lo siguiente (las zonas donde el raster está en NULL es porque no hay población):

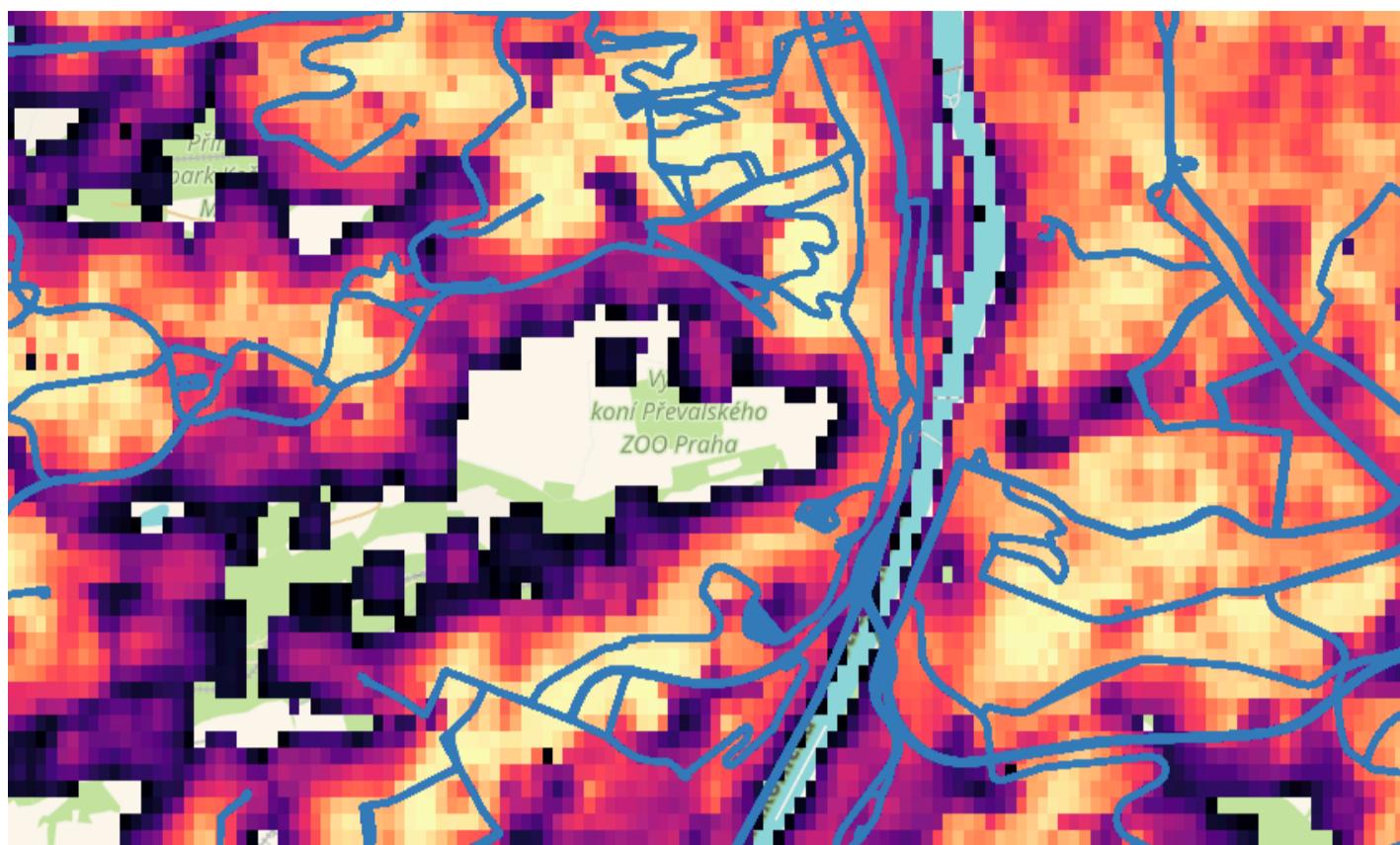


Mapa de Praga con su respectivo raster de población. Por encima, las trayectorias de los buses.

Notar como las trayectorias de los autobuses evitan zonas como bosques y zoológicos, y en el raster de población están en NULL ya que no hay población presente allí.



Focalización sobre el área forestal Kunratický les, por la que no pasan los recorridos de buses



Focalización sobre el zoológico Výběh pro koně Převalského ZOO Praha, por la que no pasan los recorridos de buses

Relación entre densidad de población y cantidad de trips que pasan por la zona en la grilla

Se utilizan índices tanto para la geometría de la ciudad como para las trayectorias de los trips, lo que agiliza mucho las operaciones como intersecciones y conteos espaciales en las consultas.

```
CREATE INDEX IF NOT EXISTS idx_province_geom ON province USING GIST (ST_Transform(geom, 5514));
CREATE INDEX IF NOT EXISTS idx_trips_mdb_traj ON trips_mdb USING GIST (traj);
```

Se crea una tabla `province_grid`, que genera una grilla de celdas cuadradas de 1 km de lado sobre un área rectangular, sin tener en cuenta los límites reales de Praga.

Esta grilla se construye utilizando los valores mínimo y máximo de longitud y latitud de la ciudad, formando así un recuadro que la engloba completamente.

```
DROP TABLE IF EXISTS province_grid CASCADE;
```

```
CREATE TABLE province_grid AS
SELECT
    row_number() OVER () AS id,
    (ST_SquareGrid(
        1000,
        ST_Transform(geom, 5514)
    )).geom
FROM province;
```

Luego, se crea otra tabla `province_grid_clipped` que toma esa grilla y recorta las celdas para conservar únicamente las partes que realmente están dentro del contorno de Praga.

```
DROP TABLE IF EXISTS province_grid_clipped CASCADE;

CREATE TABLE province_grid_clipped AS
SELECT
    g.id,
    ST_Intersection(g.geom, ST_Transform(p.geom, 5514)) AS geom
FROM province_grid g
JOIN province p
    ON ST_Intersects(g.geom, ST_Transform(p.geom, 5514))
WHERE NOT ST_IsEmpty(ST_Intersection(g.geom, ST_Transform(p.geom, 5514)));
```

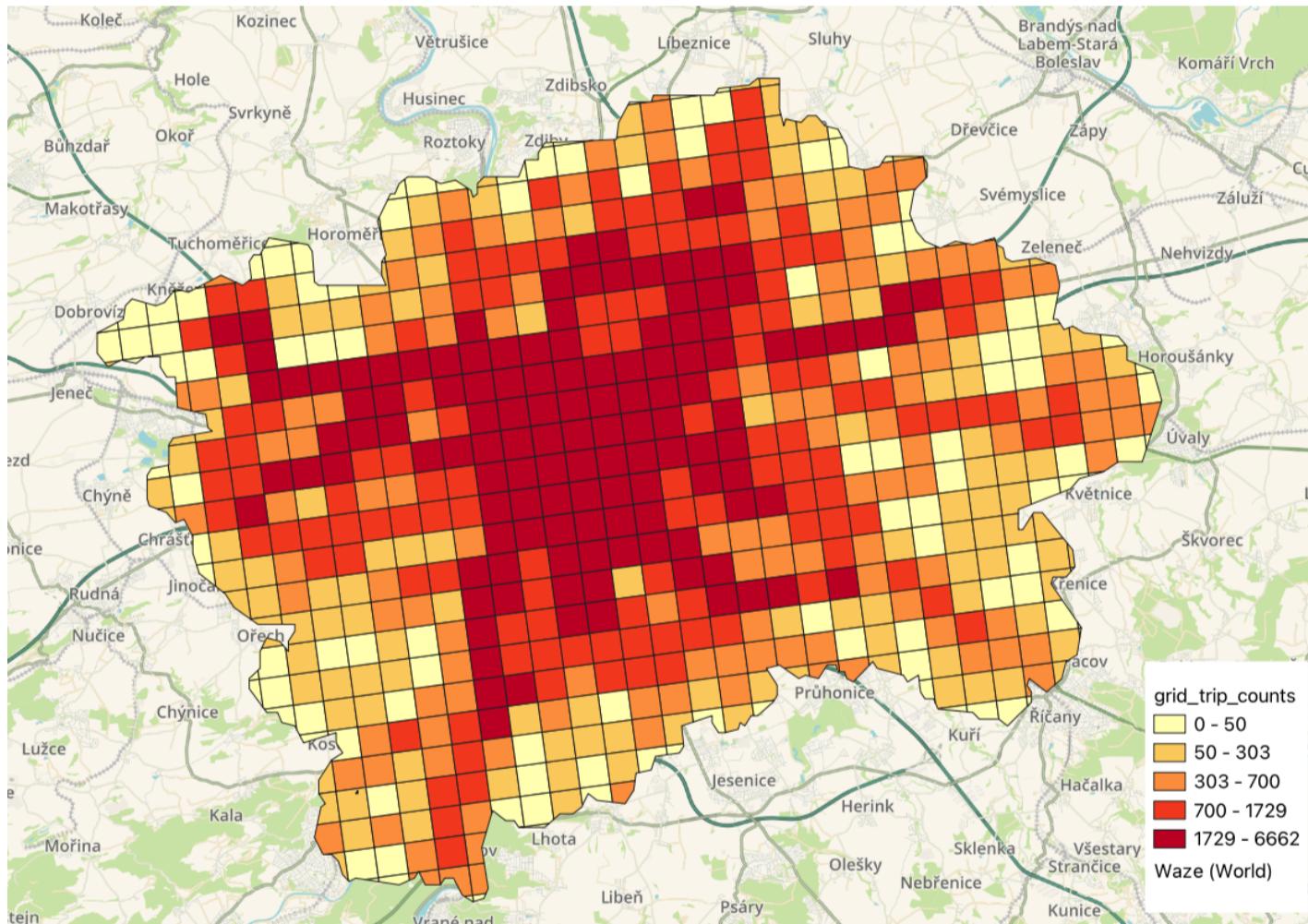
A continuación, se crea la tabla `grid_trip_counts` que cuenta la cantidad de trips que pasan por cada celda de la grilla. Así, se logrará visualizar la densidad de viajes en el territorio.

```
DROP TABLE IF EXISTS grid_trip_counts CASCADE;
```

```
CREATE TABLE grid_trip_counts AS
SELECT
    g.id AS grid_id,
    COUNT(DISTINCT t.trip_id) AS trips_count,
    g.geom
FROM province_grid_clipped g
LEFT JOIN trips_mdb t
    ON ST_Intersects(t.traj, g.geom)
GROUP BY g.id, g.geom
ORDER BY g.id;
```

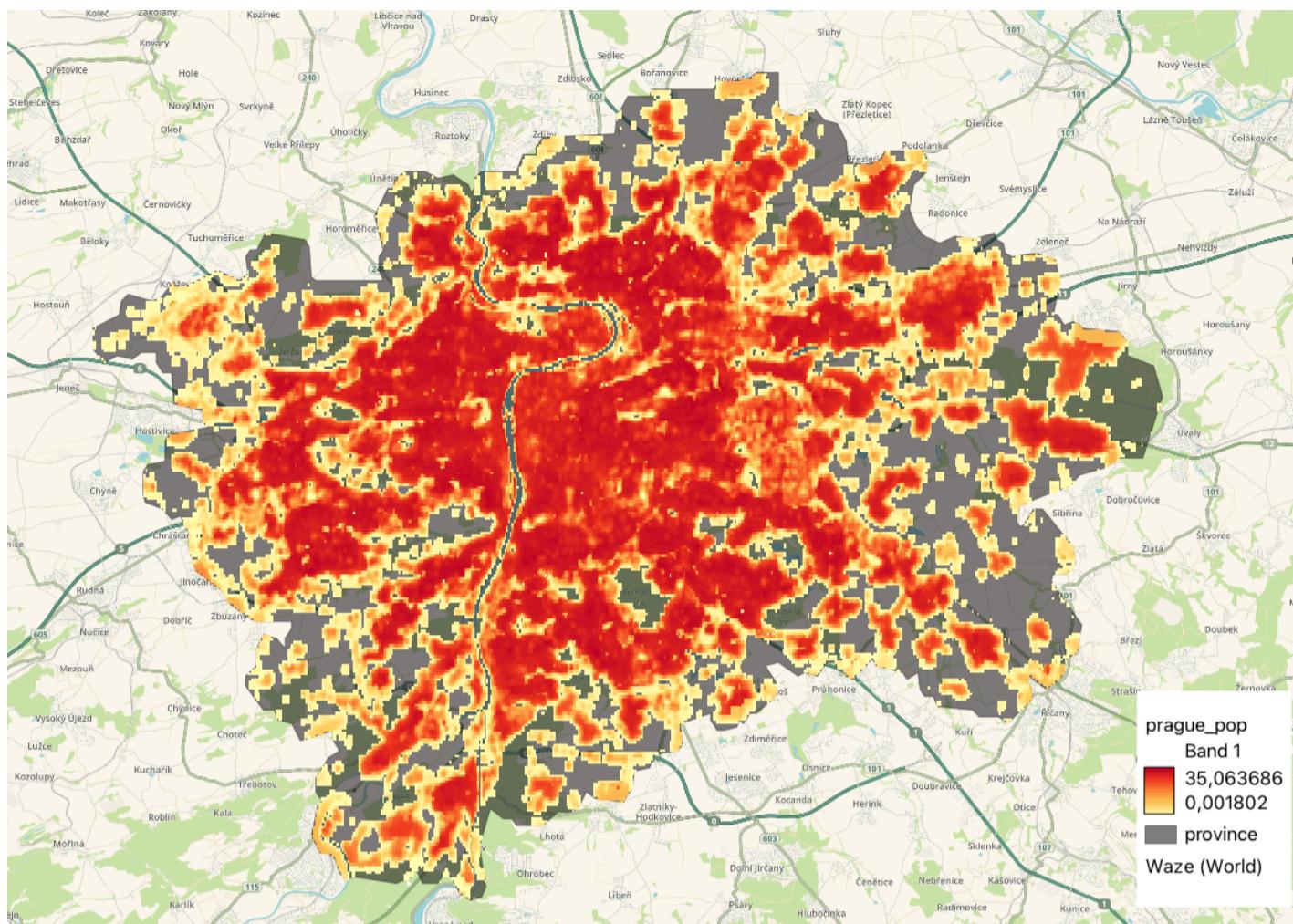
Como resultado, se puede obtener un heatmap que permite visualizar de forma rápida y clara las zonas de Praga con mayor y menor cantidad de trips registrados.

Las celdas con colores más intensos (rojos) corresponden a áreas con mayor concentración de viajes, mientras que los tonos más claros (amarillos) indican menor densidad.



Mapa de Praga con una grilla de celdas de 1km x 1km. El color de la celda depende de la cantidad de trips que pasen por ella.

Para poder observar la relación entre movilidad y población, se utilizó nuevamente el raster de población de Praga.



Raster de población de Praga

Se aplicó la misma escala de colores en el raster, permitiendo comparar visualmente las áreas con mayor y menor densidad poblacional respecto a la concentración de trayectos.

La comparación visual entre el mapa de calor de trayectos y el raster de población revela una correspondencia evidente: las zonas más transitadas de la ciudad se ubican justamente en las áreas de mayor concentración poblacional.

Esto se observa claramente en los sectores centrales y más densamente habitados de Praga.

De este modo, el análisis confirma que la demanda de movilidad está estrechamente vinculada a la distribución demográfica, ya que las zonas más pobladas son, al mismo tiempo, las que presentan mayor actividad y circulación en el sistema de transporte.

Análisis sobre la cantidad de trips que pasan por centros comerciales

Se realizará un análisis sobre la presencia de transporte público en los principales centros comerciales, con el objetivo de identificar la accesibilidad y conectividad con estos espacios.

Para comenzar, se realizó una investigación en distintos sitios (myczechrepublic.com, hotels.com) sobre los centros comerciales de Praga, identificando los más conocidos y representativos de la ciudad. A partir de esta selección, se elaboró un archivo [shoppings.geojson](#) que contiene los puntos geográficos de cada shopping, incluyendo su nombre y sus coordenadas. Las coordenadas de cada centro comercial se obtuvieron utilizando Google Maps, que emplea grados decimales en el sistema de referencia EPSG:4326.

Para importar los puntos geográficos de cada shopping a la base de datos espacial, se utilizó la herramienta [ogr2ogr](#). Este comando permitió cargar el archivo en una tabla de PostGIS, asegurando la correcta conversión de coordenadas y la creación de la geometría correspondiente.

```
ogr2ogr -f "PostgreSQL" \
PG:"host=localhost dbname=prague port=25432 user=postgres" \
shoppings.geojson \
-nln shopping_malls \
-nlt POINT \
-lco GEOMETRY_NAME=geom \
-s_srs EPSG:4326 \
-t_srs EPSG:5514 \
-overwrite
```

Por otro lado, también se investigó cuál es el centro de la ciudad de Praga y se determinó que corresponde a la Ciudad Vieja (*Staré Město*). Para representar este punto central en el análisis, se eligió un sitio turístico muy reconocido: el Reloj

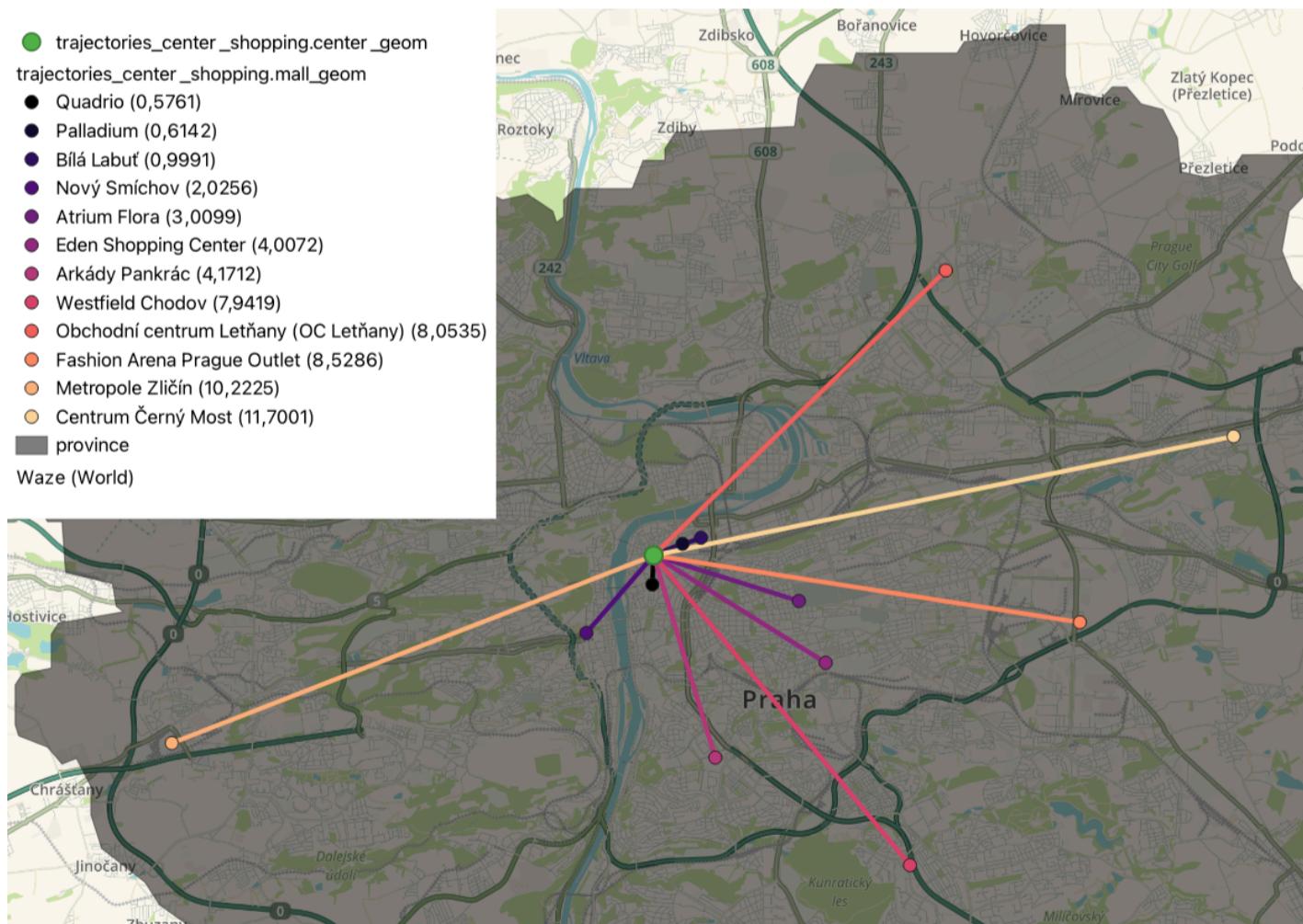
Astronómico de Praga (fuente).

Para materializar la relación espacial entre los centros comerciales y el centro de Praga, se creó la view `trajectories_center_shopping`. En ella se puede obtener el nombre de cada shopping, la ubicación del centro de la ciudad (el Reloj Astronómico), la ubicación del shopping, la distancia en línea recta entre ambos puntos (calculada en kilómetros) y la geometría de la línea que conecta ambos lugares.

```
DROP MATERIALIZED VIEW IF EXISTS trajectories_center_shopping;

CREATE MATERIALIZED VIEW trajectories_center_shopping AS
SELECT
    s.name,
    ST_SetSRID(ST_MakePoint(14.420917, 50.087008), 4326) AS center_geom,
    ST_Transform(s.geom, 4326) AS mall_geom,
    ST_DistanceSphere(
        ST_SetSRID(ST_MakePoint(14.420917, 50.087008), 4326),
        ST_Transform(s.geom, 4326)
    ) / 1000.0 AS distance_km,
    ST_MakeLine(
        ST_SetSRID(ST_MakePoint(14.420917, 50.087008), 4326),
        ST_Transform(s.geom, 4326)
    ) AS geom
FROM shopping_malls s
ORDER BY distance_km;
```

Por último, la información resultante fue visualizada en QGIS, donde se representó gráficamente la ubicación de los shoppings y su distancia en línea recta hasta el centro de la ciudad (marcado con un punto verde). Para facilitar la interpretación visual, se aplicó una simbología en la que los shoppings más cercanos al Reloj Astronómico aparecen con colores más oscuros, mientras que los más alejados se muestran con colores más claros. En la leyenda, se indica entre paréntesis la distancia en kilómetros de cada shopping al centro.



Visualización del mapa de Praga con segmentos trazados entre el Reloj Astronómico de Praga y los diferentes centros comerciales. Mientras más cerca esté el shopping, más oscuro el segmento

Se creó la tabla `shopping_trip_intervals`, que permite analizar la cantidad de trips cercanos (a 200 metros a la redonda de cada shopping) en distintos intervalos horarios de dos horas.

```

DROP TABLE IF EXISTS shopping_trip_intervals;
CREATE TABLE shopping_trip_intervals (
    shopping_name TEXT,
    interv TEXT,
    trips_nearby INTEGER
);

INSERT INTO shopping_trip_intervals (shopping_name, interv, trips_nearby)
WITH instants AS (
    SELECT
        s.name AS shopping_name,
        t.trip_id,
        getTimestamp(unnest(instants(t.trip))) AS instant_time
    FROM shopping_malls s
    JOIN trips_mdb t ON ST_DWithin(s.geom, t.traj, 200)
), ranges AS (
    SELECT
        shopping_name,
        trip_id,
        instant_time,
        (EXTRACT(HOUR FROM instant_time)::int / 2) * 2 AS range_start
    FROM instants
), ranges AS (
    SELECT
        shopping_name,
        lpad(range_start::text, 2, '0') || ':00-' || lpad((range_start+1)::text, 2, '0') || ':59' AS interv,
        COUNT(DISTINCT trip_id) AS trips_nearby
    FROM ranges
    GROUP BY shopping_name, range_start
    ORDER BY shopping_name, range_start;
)
SELECT
    shopping_name,
    lpad(range_start::text, 2, '0') || ':00-' || lpad((range_start+1)::text, 2, '0') || ':59' AS interv,
    COUNT(DISTINCT trip_id) AS trips_nearby
FROM ranges
ORDER BY shopping_name, range_start;

```

Una vez obtenida esta información, se generó el siguiente gráfico ejecutando el archivo [tips_near_shopping.py](#), en el que se puede observar que todos los shoppings presentan un comportamiento similar, con picos de viajes tanto en los horarios de apertura como en el rango de 16:00 a 18:00, coincidiendo con la salida del horario laboral.

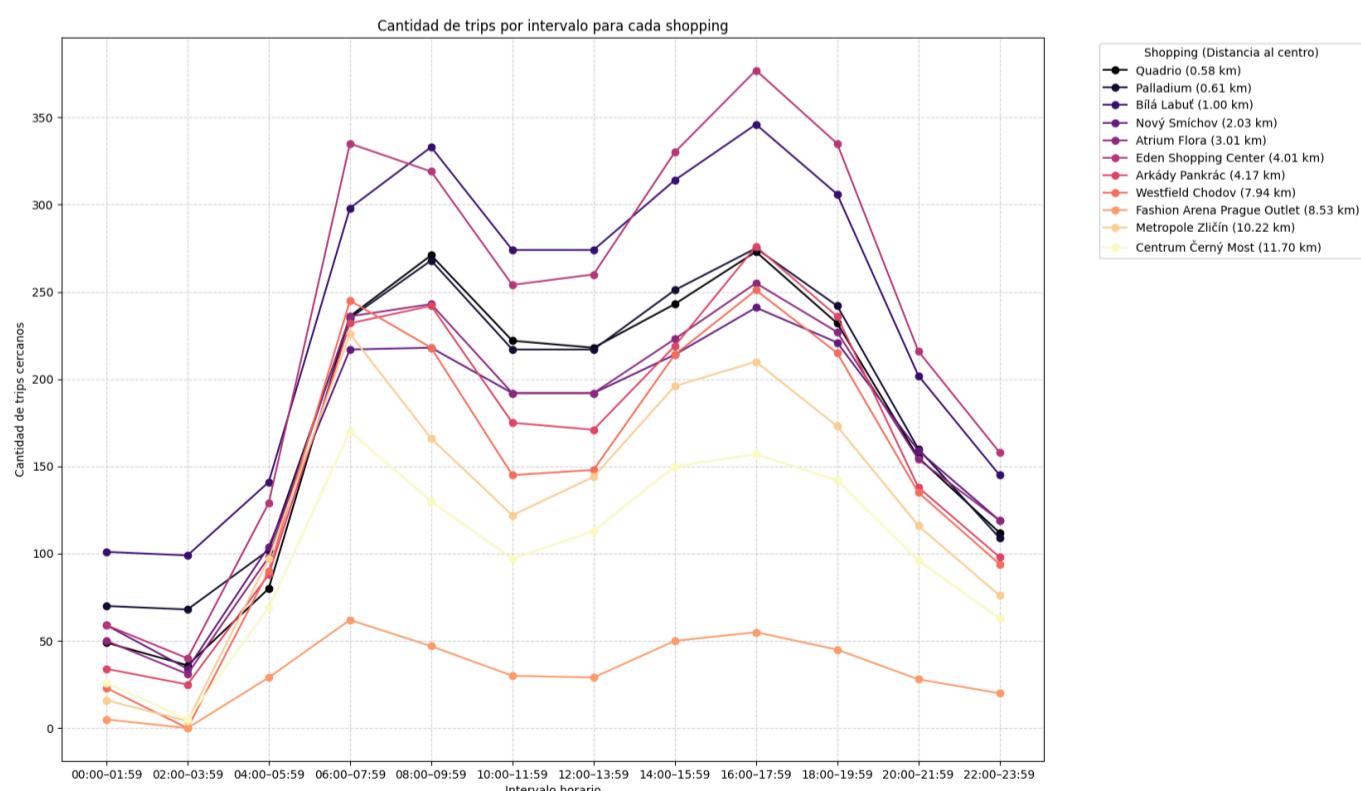


Gráfico que visualiza la cantidad de trips por intervalo de 2 horas para cada shopping

Cabe aclarar que los colores utilizados en el gráfico coinciden con los que se eligieron previamente para la visualización en QGIS, ya que se buscó que ambos mantuvieran la misma escala cromática para observar una relación de proporcionalidad inversa: a menor distancia al Reloj Astronómico, mayor es la cantidad de trips programados en las

inmediaciones del shopping a lo largo del día. En definitiva, los datos confirman una intuición urbana: la vida y el movimiento en la ciudad tienden a concentrarse cerca del centro. Así, los centros comerciales ubicados a poca distancia del corazón de Praga se convierten, naturalmente, en puntos de encuentro y destino para quienes se desplazan por la ciudad.

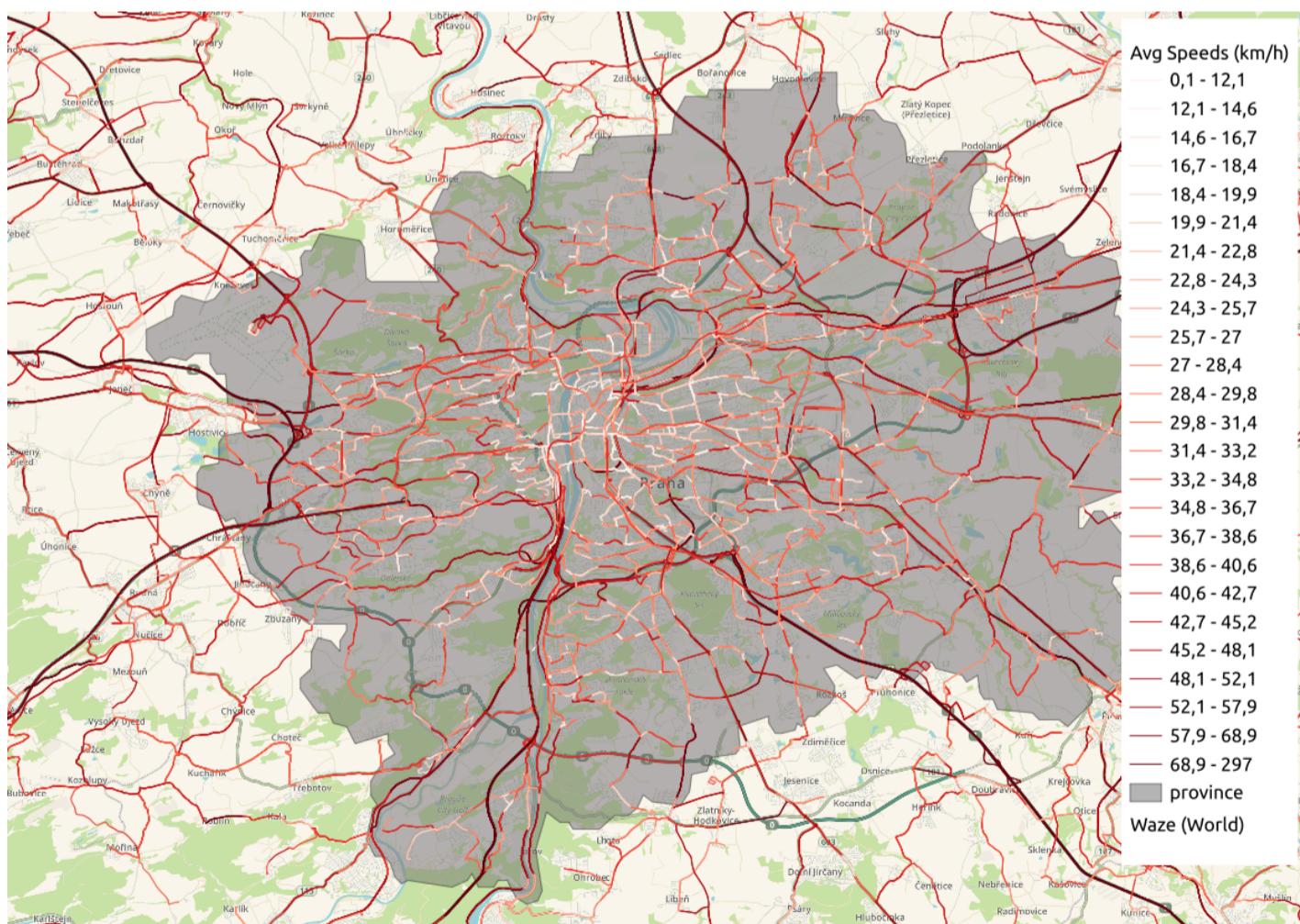
Análisis de velocidades

Se calculan las velocidades promedio cada segmento para los viajes de GTFS scheduled para el 8 de julio, y se visualizan en QGIS.

```
DROP MATERIALIZED VIEW IF EXISTS schedule_speeds;
CREATE MATERIALIZED VIEW schedule_speeds AS
SELECT
    ROW_NUMBER() OVER () as id,
    AVG(seg_length / EXTRACT(EPOCH FROM (stop2_arrival_time - stop1_arrival_time)) * 3.6) AS speed_kmh,
    seg_geom
FROM trip_segs s
WHERE stop2_arrival_time <> stop1_arrival_time
GROUP BY stop1_id, stop2_id, seg_geom;

SELECT COUNT(*) FROM trip_segs
WHERE stop2_arrival_time <> stop1_arrival_time
-- 1281030

SELECT COUNT(*) FROM trip_segs
WHERE seg_length / EXTRACT(EPOCH FROM (stop2_arrival_time - stop1_arrival_time)) * 3.6 < 30
AND stop2_arrival_time <> stop1_arrival_time
-- 821914 (el 60% de los segmentos tiene una velocidad de 30km/h o menos)
```



Mapa que visualiza los segmentos dentro de Praga y la velocidad promedio de cada uno

Se puede observar como la velocidad promedio es mayor para aquellos segmentos que salen de la ciudad.

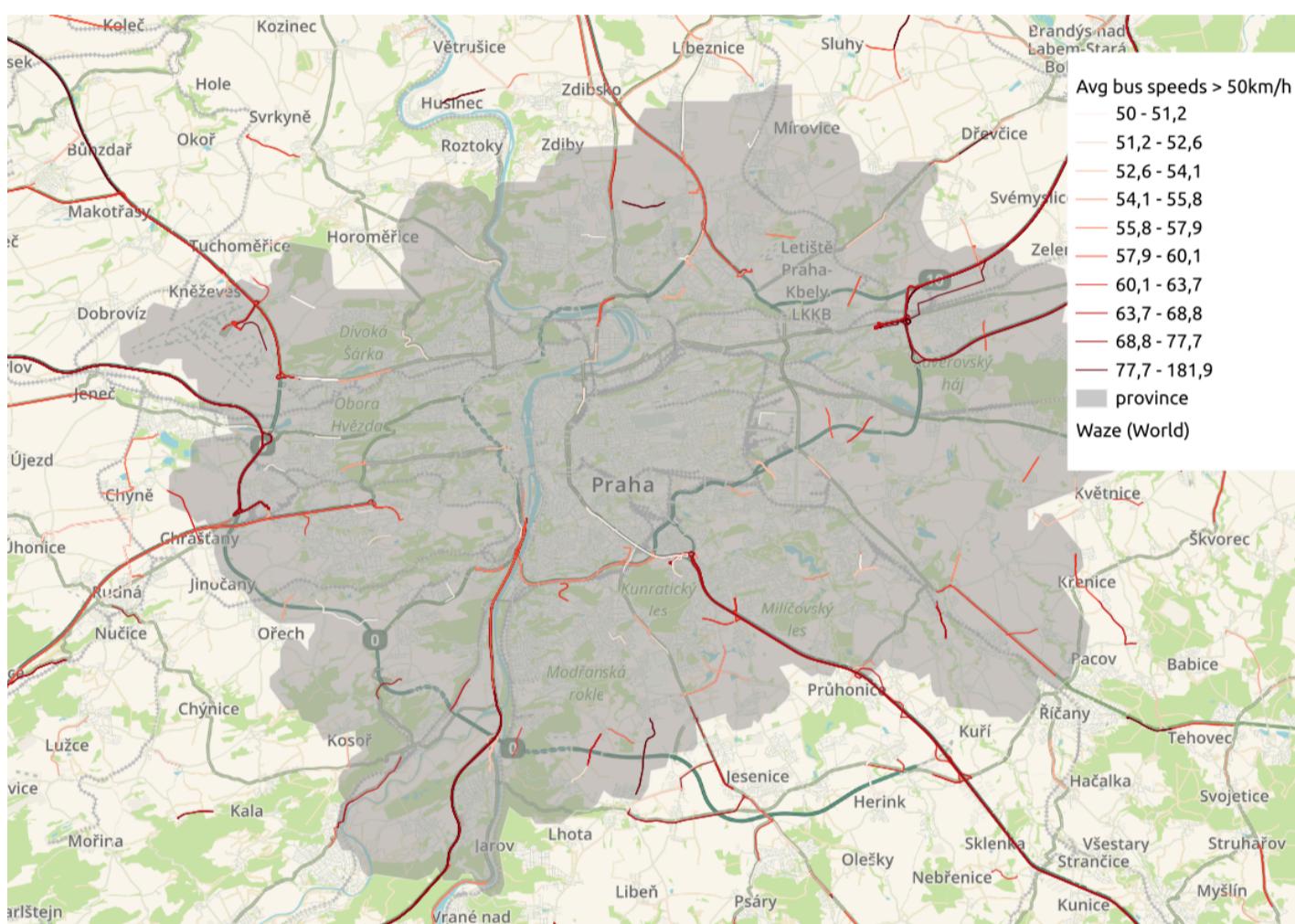
Para una mejor visualización, se pueden restringir los viajes a los segmentos de autobuses que tienen una velocidad promedio superior a 50 km/h (que es un máximo razonable para autobuses en la ciudad).

```

DROP VIEW IF EXISTS trips_over_50kmh;

CREATE MATERIALIZED VIEW trips_over_50kmh AS
SELECT
    ROW_NUMBER() OVER () AS id,
    AVG(seg_length / EXTRACT(EPOCH FROM (stop2_arrival_time - stop1_arrival_time)) * 3.6) AS speed_kmh,
    seg_geom
FROM trip_segs s
JOIN routes USING (route_id)
WHERE stop2_arrival_time <> stop1_arrival_time AND route_type = '3'
GROUP BY route_id, stop1_sequence, stop2_sequence, seg_geom
HAVING AVG(seg_length / EXTRACT(EPOCH FROM (stop2_arrival_time - stop1_arrival_time)) * 3.6) > 50;

```



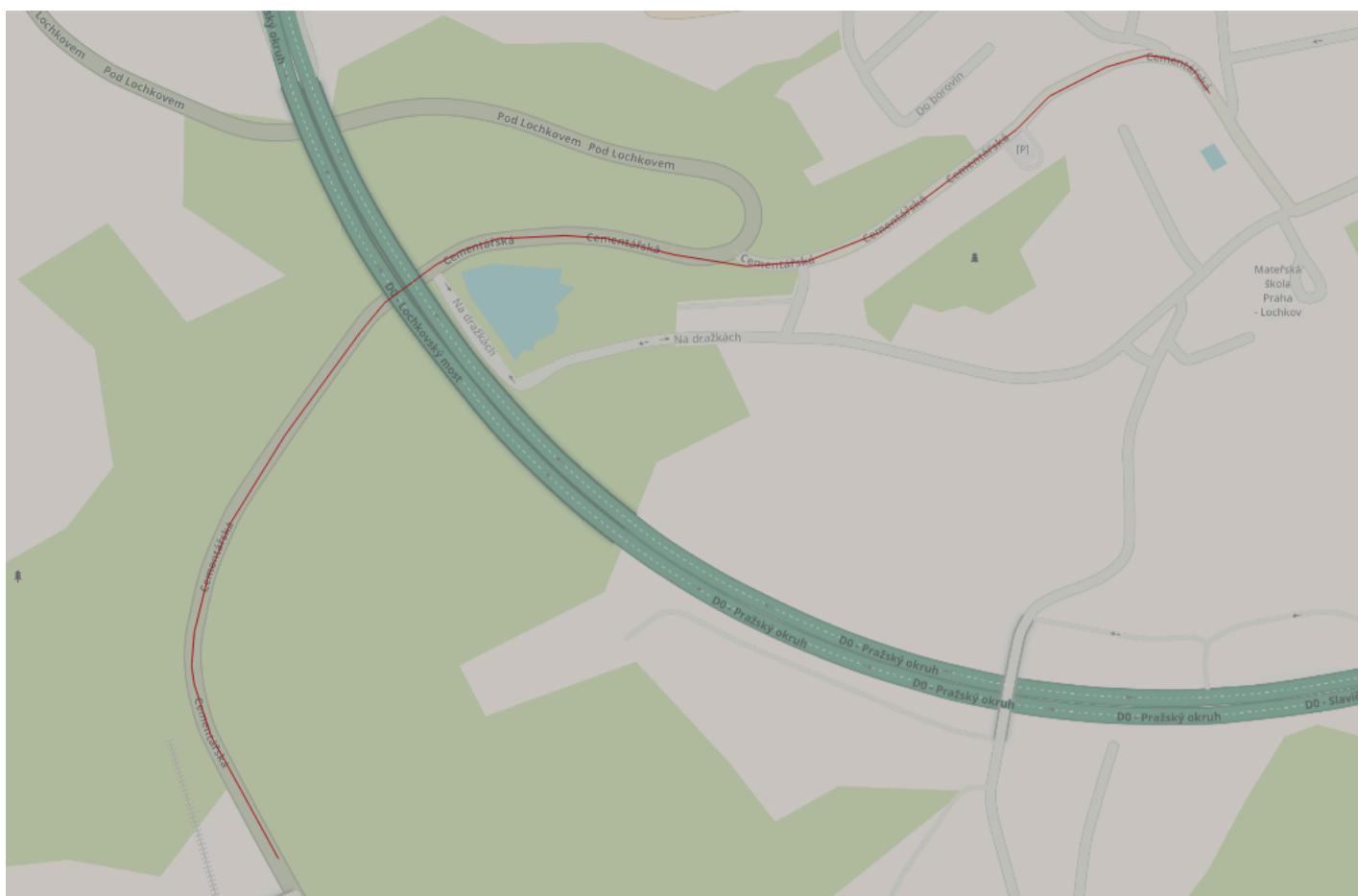
Mapa que visualiza los segmentos dentro de Praga cuya velocidad promedio es mayor a los 50 km/h

Es de interés analizar alguno de los trips que caen dentro de Praga, para ver por qué tienen tanta velocidad. Para esto, se puede hacer por ejemplo un buffer de 10 metros sobre Pod Lochkovem / Do Pražského okruhu, que se encuentra al suroeste de Praga:

```

SELECT * FROM trips_over_50kmh
WHERE ST_Intersects(ST_Transform(seg_geom, 5514), ST_Buffer(ST_Point(-749498.6,-1051660.6, 5514), 10))

```



Focalización en una de las rutas de Praga con mayor velocidad promedio

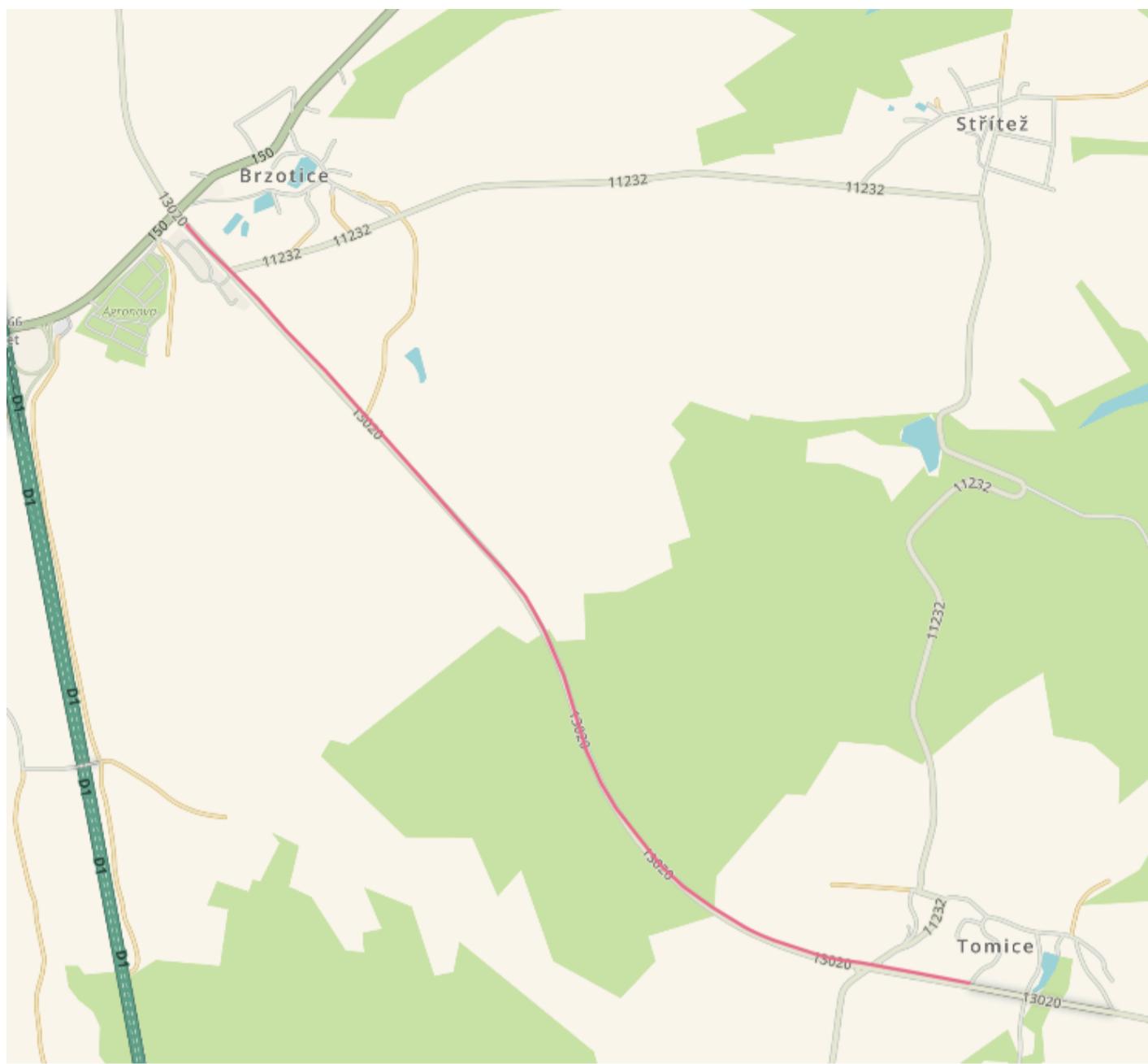
Este tramo indica que la velocidad promedio es de 76km/h, lo que viendo el mapa se podría explicar con que se trata una ruta de conexión hacia el anillo de circunvalación de Praga (Pražský okruh). Esto implica que tiene características de autopista: no tiene semáforos, ni cruces peatonales.

Outliers con alta velocidad

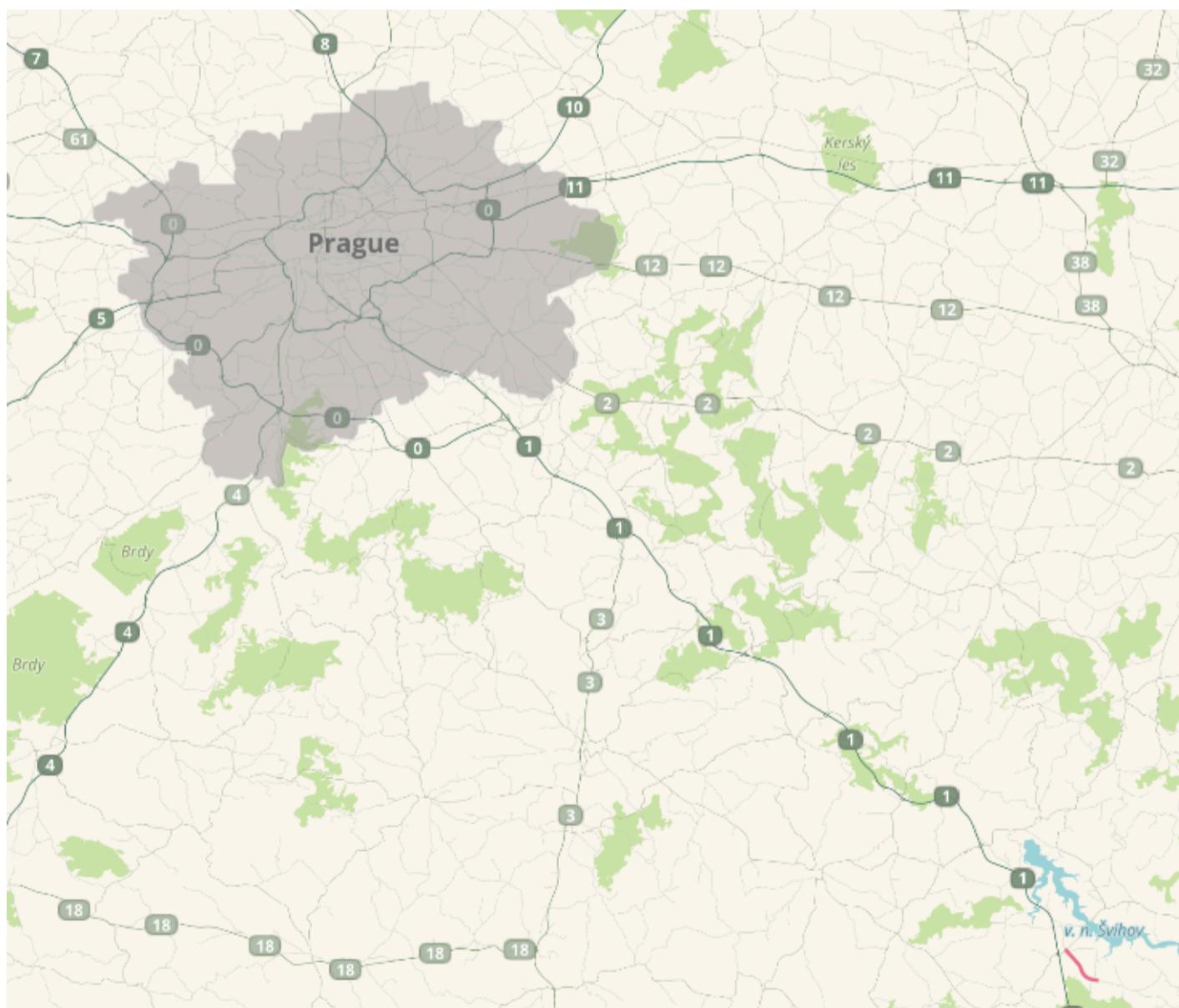
En el mapa de velocidades se veía que había buses que alcanzan velocidades de hasta 180km/h.

Por ejemplo para el máximo, se observa que se trata de un bus que viaja por la ruta 13020, bastante lejos de Praga, en un segmento con varios tramos rectos, lo que explica perfectamente su velocidad.

```
SELECT * FROM trips_over_50kmh
ORDER BY speed_kmh DESC
LIMIT 1
```



Focalización en el segmento con máxima velocidad promedio



Alejamiento del mapa de Praga para visualizar la lejanía de Praga de la ubicación del segmento con máxima velocidad promedio

Análisis de velocidades por barrio

Para comenzar el análisis de las velocidades por barrio, se obtuvo un raster de Praga dividido en sus diferentes distritos. Este conjunto de datos espaciales fue extraído del portal oficial de datos abiertos de la República Checa.

Los datos fueron descargados y guardados en el archivo [prague_districts22.geojson](#), el cual se encuentra disponible en la carpeta de Drive.

Para importar y gestionar los datos geoespaciales de los distritos dentro de la base de datos, se utilizó, al igual que en otras queries, [ogr2ogr](#).

El siguiente comando fue empleado para importar los datos del archivo en la base de datos:

```
ogr2ogr -f "PostgreSQL" \
PG:"host=localhost dbname=prague port=25432 user=postgres" \
prague_districts22.geojson \
-nln prague_districts \
-nlt MULTIPOLYGON \
-lco GEOMETRY_NAME=geom \
-overwrite
```

Se realizan las siguientes consultas para analizar la velocidad promedio del transporte público en los diferentes distritos de Praga durante diferentes intervalos horarios de 6 horas.

```
DROP MATERIALIZED VIEW IF EXISTS avg_speed_by_district_0_6;
CREATE MATERIALIZED VIEW avg_speed_by_district_0_6 AS
SELECT
    d.naz_sop AS district,
    d.geom AS district_geom,
    AVG(s(seg_length / EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) * 3.6) AS avg_speed_kmh
FROM trip_segs s
JOIN prague_districts d
    ON ST_Intersects(s.seg_geom, d.geom)
WHERE
    s.stop2_arrival_time <> s.stop1_arrival_time
    AND EXTRACT(HOUR FROM s.stop1_arrival_time) >= 0
    AND EXTRACT(HOUR FROM s.stop1_arrival_time) < 6
GROUP BY d.naz_sop, d.geom
ORDER BY d.naz_sop;

DROP MATERIALIZED VIEW IF EXISTS avg_speed_by_district_6_12;
CREATE MATERIALIZED VIEW avg_speed_by_district_6_12 AS
SELECT
    d.naz_sop AS district,
    d.geom AS district_geom,
    AVG(s(seg_length / EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) * 3.6) AS avg_speed_kmh,
    COUNT(*) AS segment_count
FROM trip_segs s
JOIN prague_districts d
    ON ST_Intersects(s.seg_geom, d.geom)
WHERE
    s.stop2_arrival_time <> s.stop1_arrival_time
    AND EXTRACT(HOUR FROM s.stop1_arrival_time) >= 6
    AND EXTRACT(HOUR FROM s.stop1_arrival_time) < 12
GROUP BY d.naz_sop, d.geom
ORDER BY d.naz_sop;

DROP MATERIALIZED VIEW IF EXISTS avg_speed_by_district_12_18;
CREATE MATERIALIZED VIEW avg_speed_by_district_12_18 AS
SELECT
    d.naz_sop AS district,
    d.geom AS district_geom,
```

```

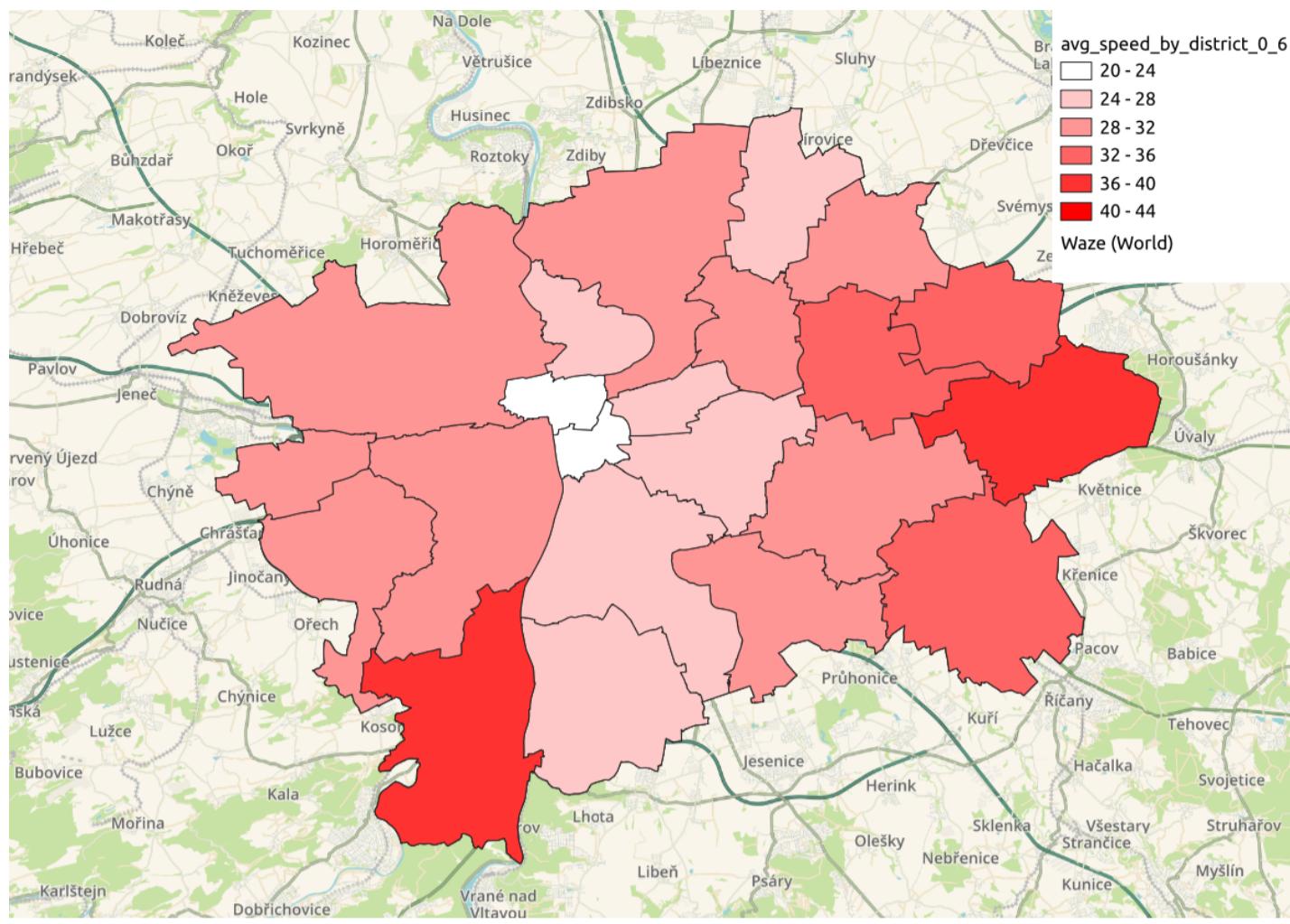
AVG(s(seg_length / EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) * 3.6) AS avg_speed_kmh,
COUNT(*) AS segment_count
FROM trip_segs s
JOIN prague_districts d
ON ST_Intersects(s.seg_geom, d.geom)
WHERE
s.stop2_arrival_time <> s.stop1_arrival_time
AND EXTRACT(HOUR FROM s.stop1_arrival_time) >= 12
AND EXTRACT(HOUR FROM s.stop1_arrival_time) < 18
GROUP BY d.naz_sop, d.geom
ORDER BY d.naz_sop;

DROP MATERIALIZED VIEW IF EXISTS avg_speed_by_district_18_24;
CREATE MATERIALIZED VIEW avg_speed_by_district_18_24 AS
SELECT
d.naz_sop AS district,
d.geom AS district_geom,
AVG(s(seg_length / EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) * 3.6) AS avg_speed_kmh,
COUNT(*) AS segment_count
FROM trip_segs s
JOIN prague_districts d
ON ST_Intersects(s.seg_geom, d.geom)
WHERE
s.stop2_arrival_time <> s.stop1_arrival_time
AND EXTRACT(HOUR FROM s.stop1_arrival_time) >= 18
AND EXTRACT(HOUR FROM s.stop1_arrival_time) < 24
GROUP BY d.naz_sop, d.geom
ORDER BY d.naz_sop;

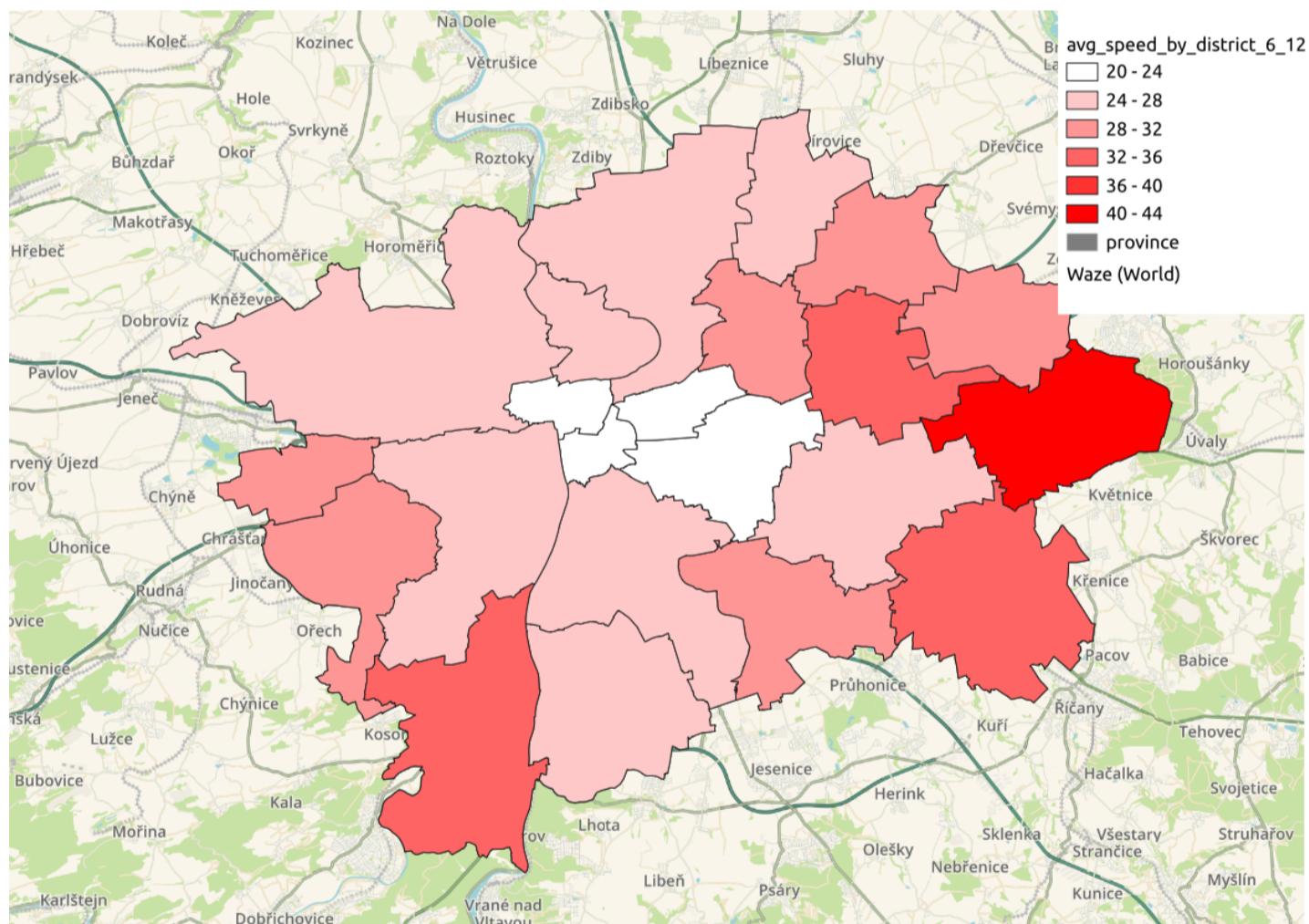
```



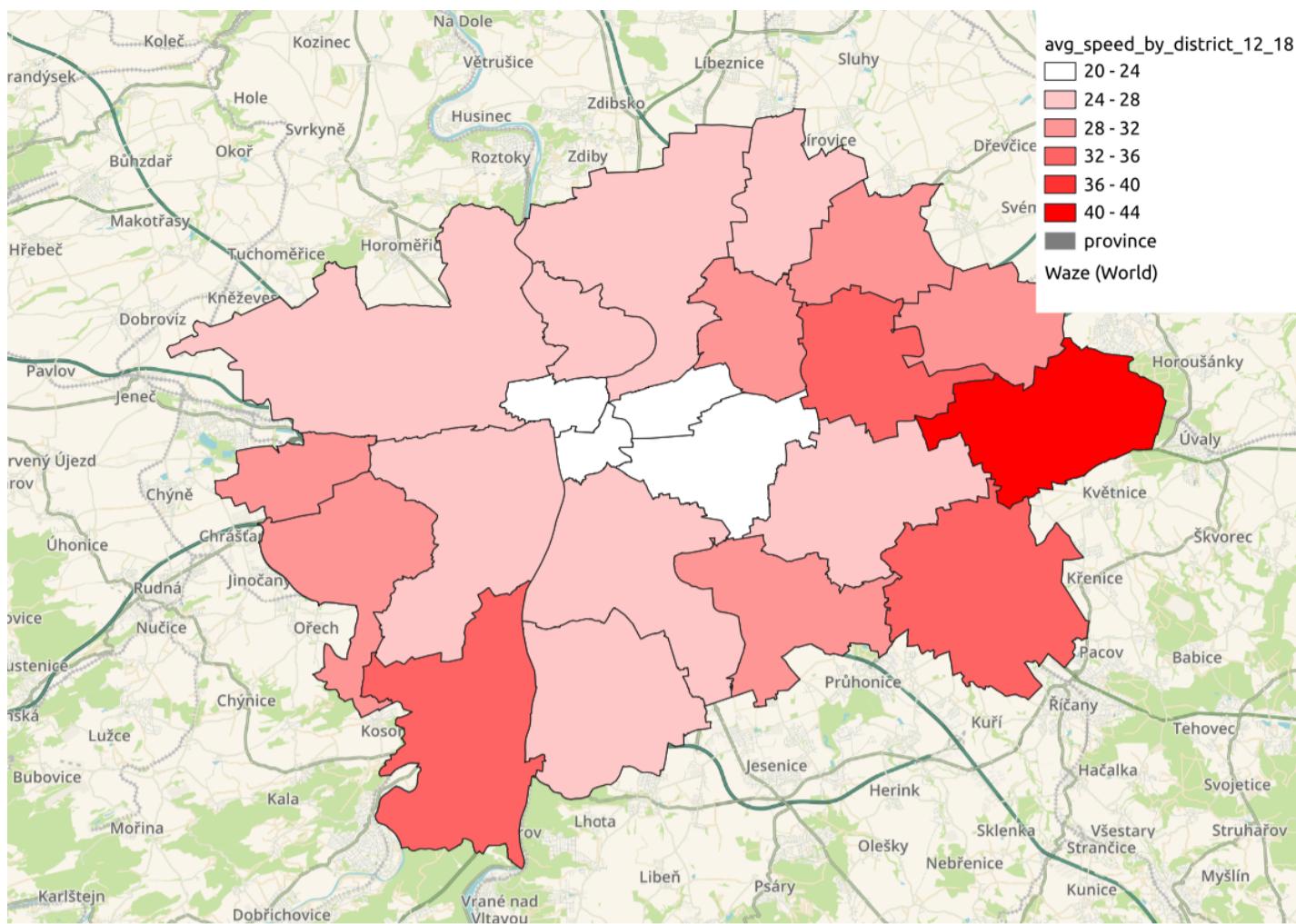
Distritos de praga (https://en.wikipedia.org/wiki/Districts_of_Prague#/media/File:Prague_Subdivisions.svg)



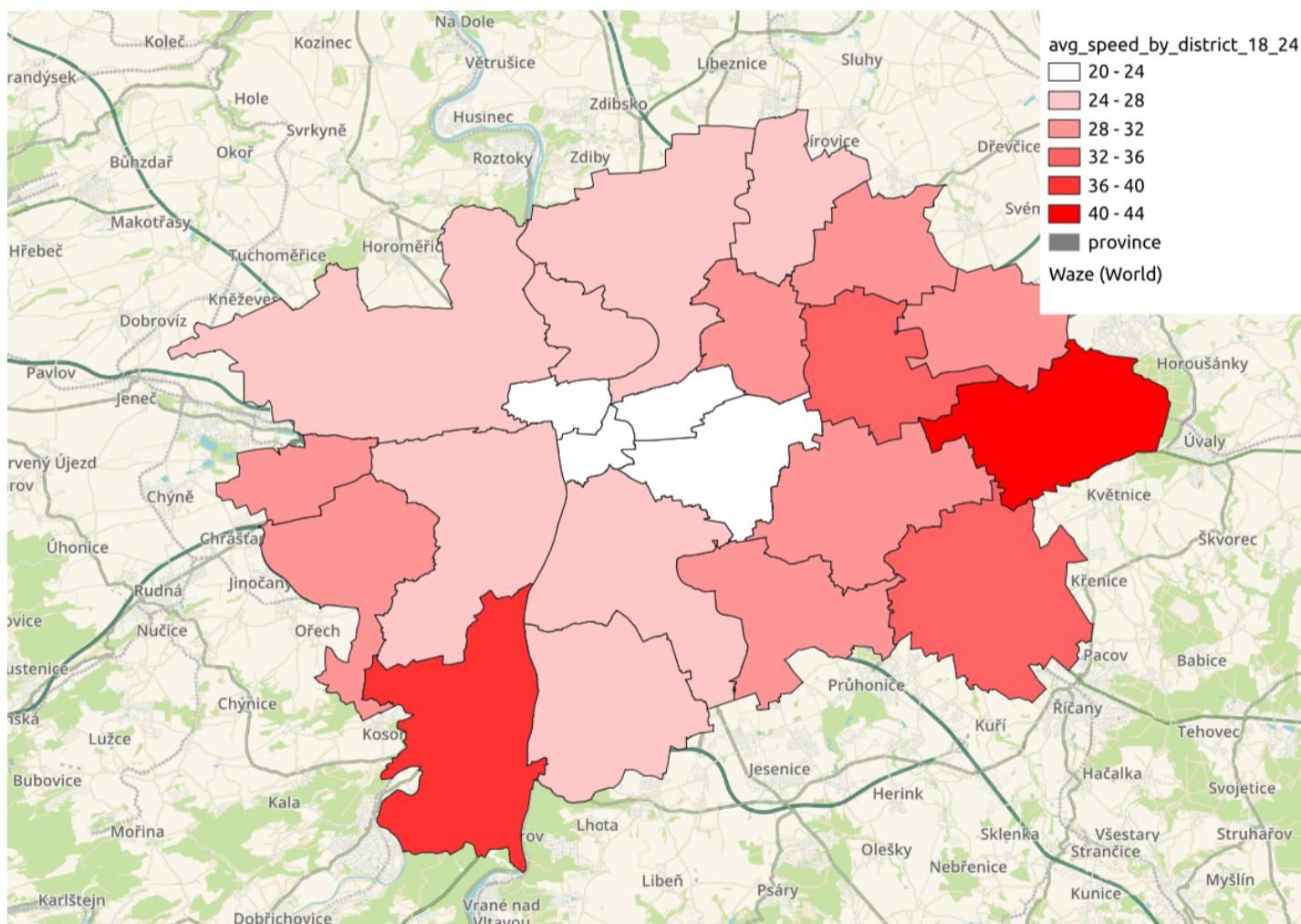
Velocidad promedio (km/h) por distrito de 0hs a 6hs



Velocidad promedio (km/h) por distrito de 6hs a 12hs



Velocidad promedio (km/h) por distrito de 12hs a 18hs (aclaración: la velocidad mínima es 19.95, pero se redondeo a 20)



Velocidad promedio (km/h) por distrito de 18hs a 24hs

Durante la madrugada (0 a 6 hs), las velocidades son considerablemente más altas, superando los 40 km/h en distritos periféricos como Praha 16, Praha 21 o Praha 22. Esto puede deberse, en parte, a la baja densidad de tránsito en esas horas y al diseño más abierto de sus calles. En contraste, el centro histórico (Praha 1 y Praha 2) muestra velocidades más bajas, posiblemente debido a su estructura urbana más cerrada y antigua. Este comportamiento sugiere que hay espacio para repensar cómo se organiza el tránsito en zonas centrales fuera del horario pico.

En la mañana (6 a 12 hs) y la tarde (12 a 18 hs), se observa una caída en las velocidades promedio en todos los distritos. Las zonas más afectadas son las centrales y semi-centrales como Praha 3, Praha 5, y Praha 10, donde se suelen realizar actividades laborales y escolares. Este momento del día podría aprovecharse para realizar ajustes dinámicos en la frecuencia o la sincronización de los semáforos, mejorando la eficiencia del sistema en tiempo real.

Hacia la noche (18 a 24 h), la velocidad vuelve a aumentar. Zonas como Praha 15 y Praha 16 muestran valores que se acercan nuevamente a los observados durante la madrugada. Sin embargo, el centro mantiene velocidades relativamente

bajas.

GTFS Real Time

Como se mencionó, se decidió utilizar la [API Rest de golemio](#) para la obtención de los datos GTFS real time.

Para esto se utilizaron varios scripts de python:

rest_gtfs_rt_inspector

Obtiene los datos de las posiciones de los vehículos de la API REST, en tiempo real. Hace polling cada 20 segundos y puede cortarse con un tiempo limite o al mandar un SIGINT con Ctrl + C.

Un ejemplo de una respuesta de la API podría ser el siguiente.

```
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          14.441252,
          50.109318
        ],
        "type": "Point"
      },
      "properties": {
        "gtfs_trip_id": "705_735_231002",
        "route_type": "bus",
        "gtfs_route_short_name": "705",
        "bearing": 221,
        "delay": 20,
        "vehicle_id": "service-3-8585",
        "state_position": "at_stop"
      },
      "type": "Feature"
    }
  ],
  "type": "FeatureCollection"
}
```

A partir de esta información, se arma una lista de objetos con los siguientes campos:

- `longitude`
- `latitude`
- `vehicle_id`
- `trip_id`
- `route_id` : es el `route_short_name` de `routes` en schedule
- `startdate`
- `timestamp`

Cuando termina de recopilar la información, elimina los registros que son exactamente iguales salvo por el valor de timestamp. Esto se hizo así ya que como se puede ver en el ejemplo, la respuesta no incluye el timestamp de los datos, por lo que es imposible determinar si un punto que llegó igual en dos momentos distintos fue por un problema de precisión del GPS, o porque el vehículo se mantuvo quieto.

El resultado final, se guarda en un archivo parquet.

map_matching

Utiliza Valhalla Meili para hacer map matching sobre los trips del archivo parquet.

Para esto se decidió levantar un servidor de Valhalla local utilizando docker, con el siguiente comando.

```
docker run -dt --name valhalla_gis-ops -p 8002:8002 -v $PWD/custom_files:/custom_files -e tile_urls=https://download.geofabrik.de/europe/czech-republic-latest.osm.pbf ghcr.io/nilsnolde/docker-valhalla/valhalla:latest
```

También se podría utilizar el servicio público de Valhalla de https://valhalla1.openstreetmap.de/trace_route, pero lógicamente con mucha menor performance.

El proceso primero ordena por trip y timestamp, luego sobre cada trip realiza una limpieza de los datos, eliminando puntos duplicados si quedaran, y detectando inconsistencias si un vehículo se encontrara en dos lugares al mismo tiempo.

Luego, utiliza el haversine distance de `pyproj`, para descartar puntos consecutivos que se encuentran a más de 2km de distancia.

Una vez que ya tiene definidos los puntos de los trips (`lon`, `lat` y `time`), estos se envían al endpoint de `/trace_route` de Valhalla. Algunos parámetros relevantes que se pasan son:

- `costing: "auto"` : costo estándar para rutas en automóvil, motocicleta, camión, etc., que respetan las reglas de conducción de vehículos, como restricciones de acceso y de giro. El modo “auto” proporciona una ruta de tiempo corto (aunque no siempre garantiza el menor tiempo posible) y aplica un costo adicional en las intersecciones para minimizar los giros, maniobras o cambios de nombre de calle. Además, tiende a priorizar autopistas y caminos de mayor jerarquía, como autopistas principales y rutas troncales.
- `shape_match: "map_snap"`
 - `map_snap`: indica que debe utilizarse un algoritmo de map-matching, ya que la forma de entrada podría no coincidir exactamente con los segmentos de la red de Valhalla. Este algoritmo es más costoso en términos de procesamiento. Otra opción podría ser `edge_walk`, que requiere que la shape de entrada coincida casi exactamente con los segmentos de la red.
- `use_timestamps": True` : indica si se deben utilizar los timestamps o las duraciones proporcionadas en la entrada para calcular el tiempo transcurrido en cada tramo del recorrido ajustado. Si es true, se utiliza el campo time que se pasó como entrada. Si es false, se utiliza el cálculo interno del sistema para estimar los tiempos transcurridos.
- `format: "osrm"`
- `trace_options: { search_radius: 100 }`
 - `search_radius`: sirve para especificar el radio de búsqueda (en metros) dentro del cual se buscarán los caminos candidatos para cada medición. El radio de búsqueda máximo es de 100 metros. Al aumentar el valor de este radio, el rendimiento puede disminuir.

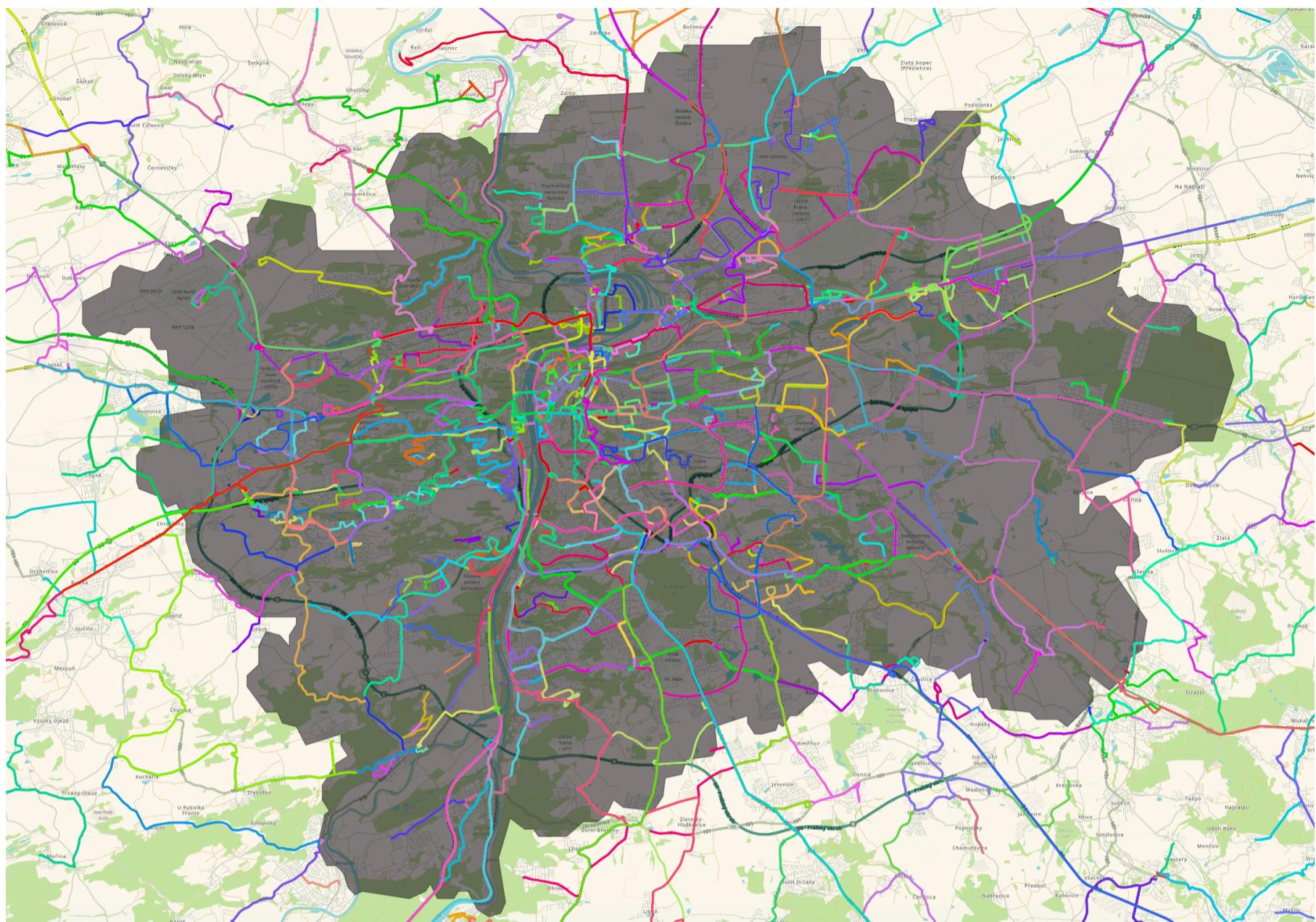
Los datos obtenidos son:

- `geometry` que representa la trayectoria map matcheada (incluyendo puntos interpolados).
- `tracepoints` que trae los mismos puntos enviados, map matcheados. En caso de que no pueda realizar el map matching, devolverá un None.

Un gran problema de este formato de respuesta es que los puntos provistos por el `geometry` no vienen junto a los timestamps interpolados por Valhalla. Se investigó si había forma de hacer que los trajera (a través del endpoint de `/trace_attributes`), pero la API no demostró ser lo suficientemente consistente. Más adelante se propondrá una solución tomando los puntos de la geometría uno a uno y agregándoles el tiempo interpolado en SQL.

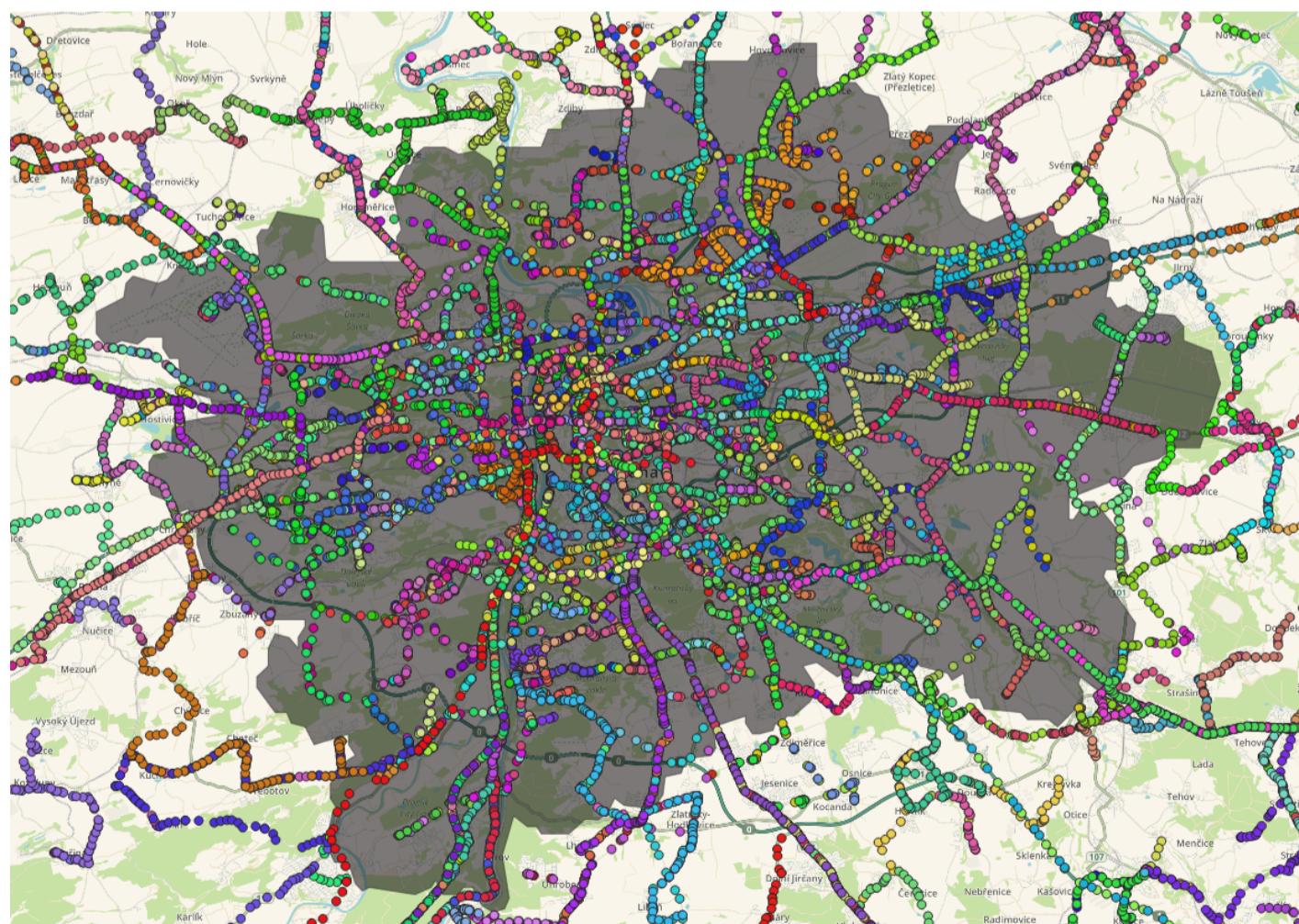
El resultado de todo este proceso son varios dataframes (que se convierten a los archivos [map_matched_positions.csv](#), [map_matched_shapes.csv](#), [map_matched_shapes.geojson](#), [map_matched_trips.geojson](#), [map_matching_errors.geojson](#), todos disponibles en la [carpeta de drive](#))

- `matched_df` : GeoDataFrame que guarda los puntos map matcheados de cada viaje. A partir de él se crea un [map_matched_trips.geojson](#) para poder visualizar todas las trayectorias atemporales sin interpolación.



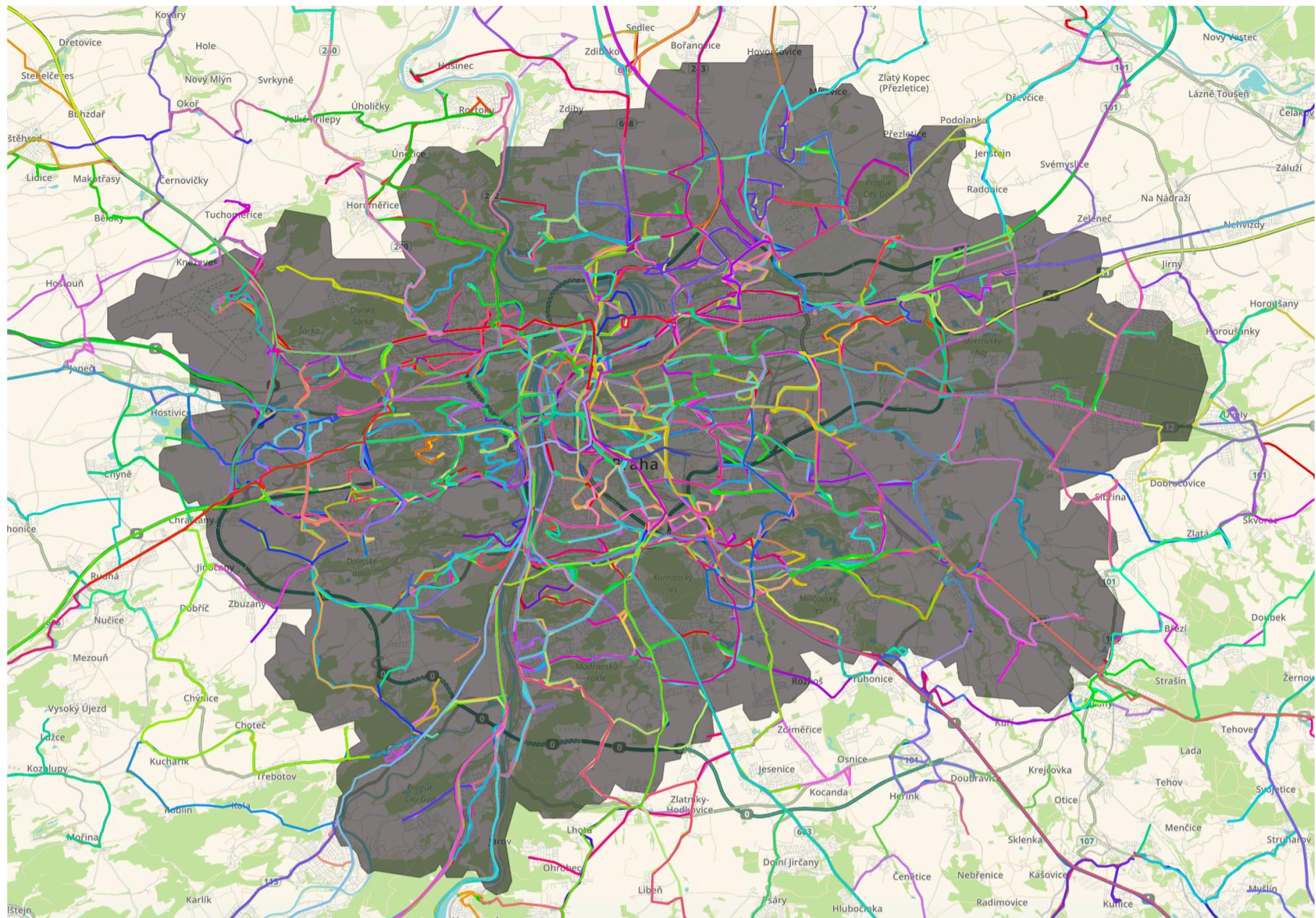
Trayectorias armadas uniendo los tracepoints retornados por Valhalla (matched_df). Notar como no siempre se respetan los caminos reales por falta de interpolación.

- `point_df` : DataFrame que tiene todo lo necesario para luego crear las tablas en sql. Produce un archivo [map_matched_positions.csv](#)



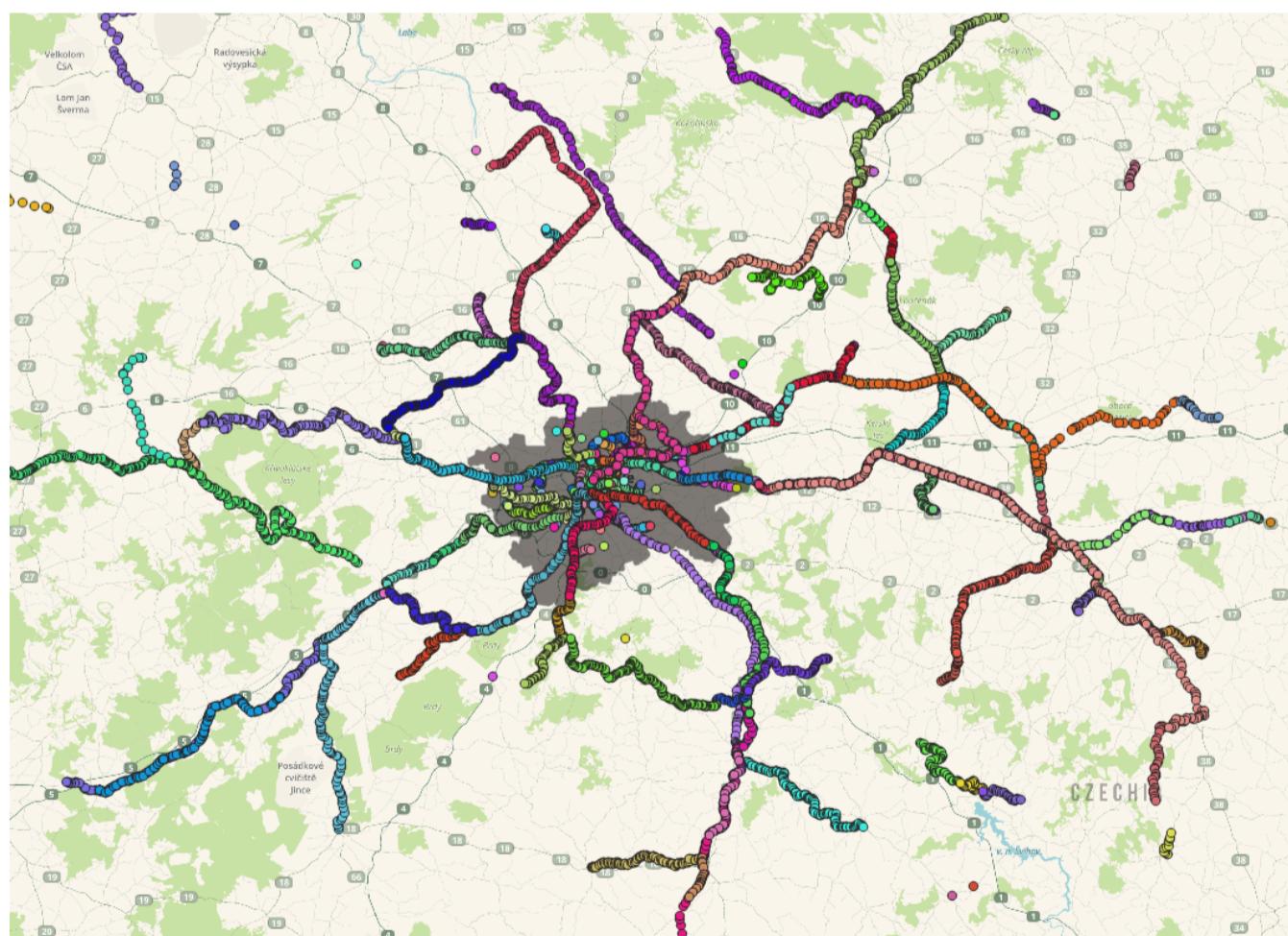
Puntos luego del map matching (point_df)

- `shapes_df` : GeoDataFrame que guarda las trayectorias retornadas por Valhalla para cada viaje. A partir de él se crea un [map_matched_shapes.geojson](#) para poder visualizar todas las trayectorias atemporales y un [map_matched_shapes.csv](#) para poder importarlas a sql.



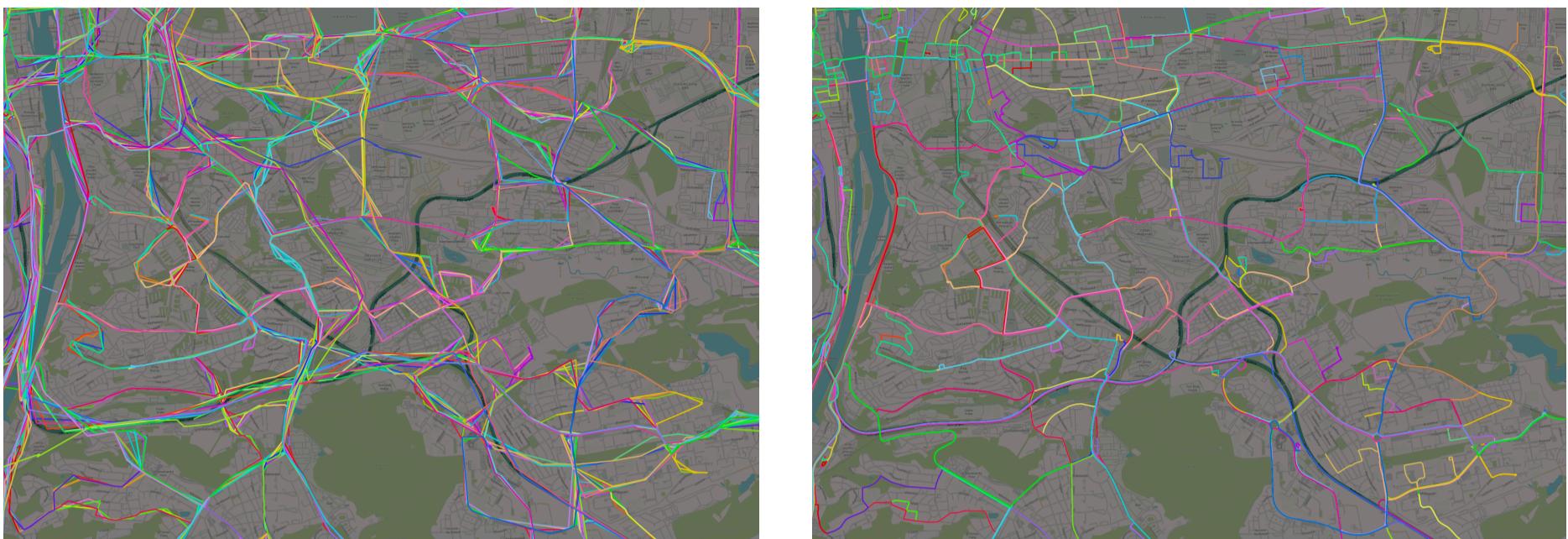
Trayectorias retornadas por Valhalla para cada viaje (shapes_df)

- failed_gdf : GeoDataFrame que guarda todas las geometrías que se descartaron por algún error.



Puntos de failed_gdf (descartados durante el proceso de map matching)

A continuación se muestra una comparativa entre matched_df (izquierda) y shapes_df (derecha). Observar como los segmentos del primero son irregulares por la falta de interpolación.



gtfs_rt_inspector

Versión alternativa del extractor de datos real time pero para la API de protobuf. También genera un parquet con las posiciones, pero debe ser preprocesado con la función `convert_parquet` del archivo `visualize.py`.

Se recomienda no utilizarlo debido a que la calidad de sus datos es bastante peor.

Análisis de trayectorias fallidas

Analizando el archivo con el script `errors.py` (teniendo la precaución de modificarlo para que apunte a la base de datos), se pueden obtener las cantidades de cada tipo de vehículo que falló:

```
python3 errors.py map_matching_errors.geojson
Fetching route types from database...

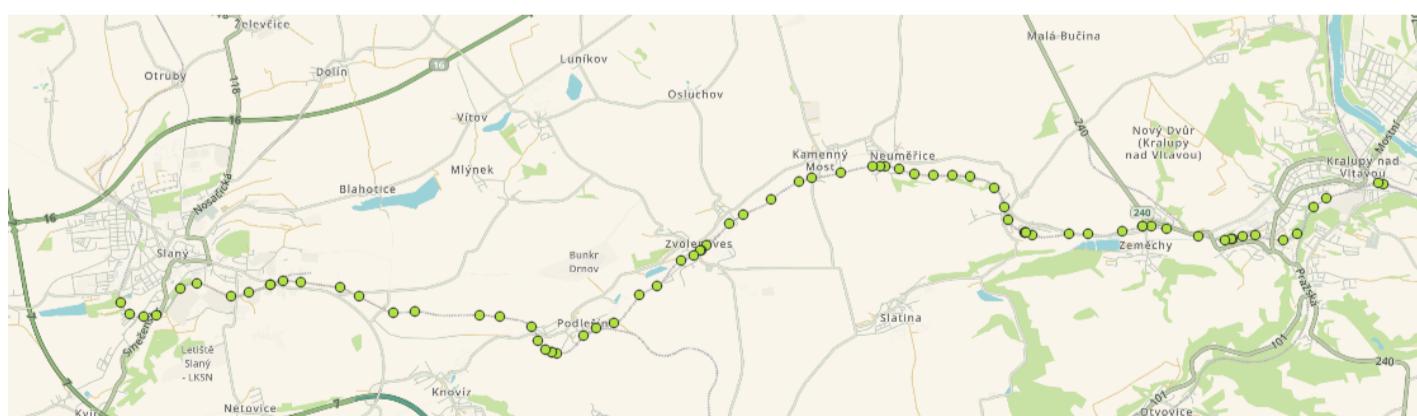
Vehicle Counts by Route Type:
Rail: 181
Bus: 49
Tram/Streetcar/Light Rail: 32
Subway/Metro: 21
Ferry: 16
Trolleybus: 2
Unknown: 9

Total vehicles: 323

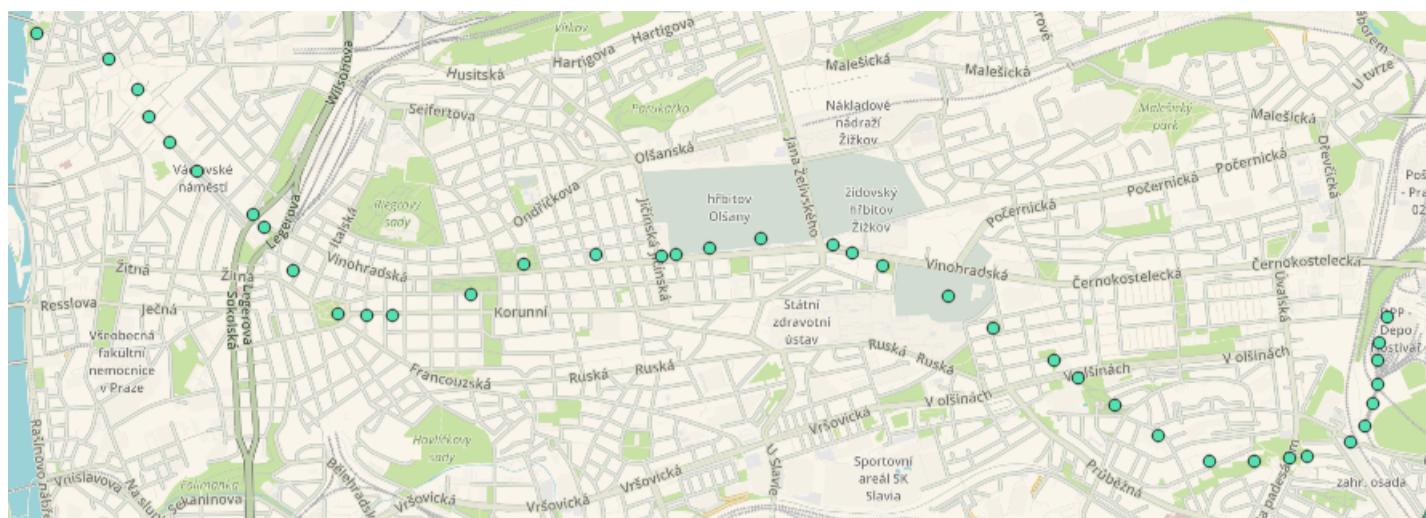
Unknown route types:
Unknown trip: {"trip_id": "170_2999_250708", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "175_3421_250708", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "177_3854_250708", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "195_4709_250708", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "213_1132_250708", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "332_1227_250707", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "339_200_250707", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "538_209_250709", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
Unknown trip: {"trip_id": "9_23499_250707", "error_code": "InsufficientPoints", "error_msg": "Less than 2 points for map matching"}
```

Aquellos trips que no figuraban en la información scheduled y tampoco empiezan en su vehicle id con el prefijo "train" o "metro", se puede ver que fallan por un tema de puntos insuficientes como para hacer el map matching.

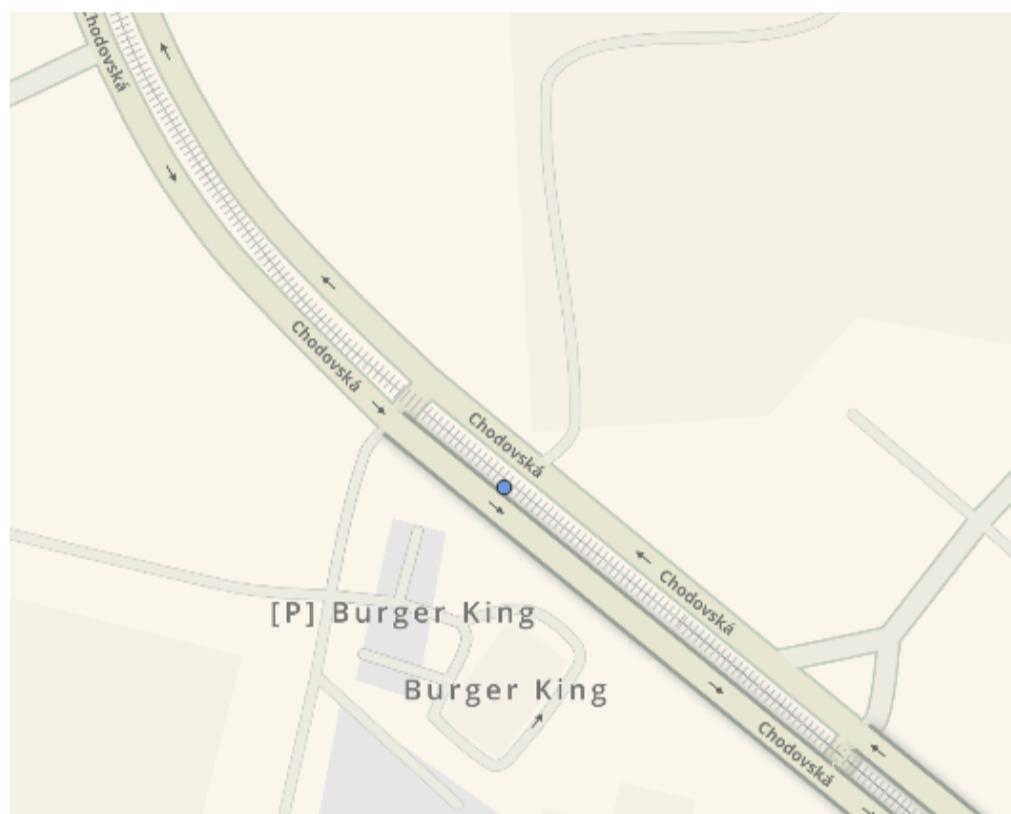
Se procede a mostrar distintos ejemplos de trips (en `map_matching_errors.geojson`) que no se pudieron procesar:



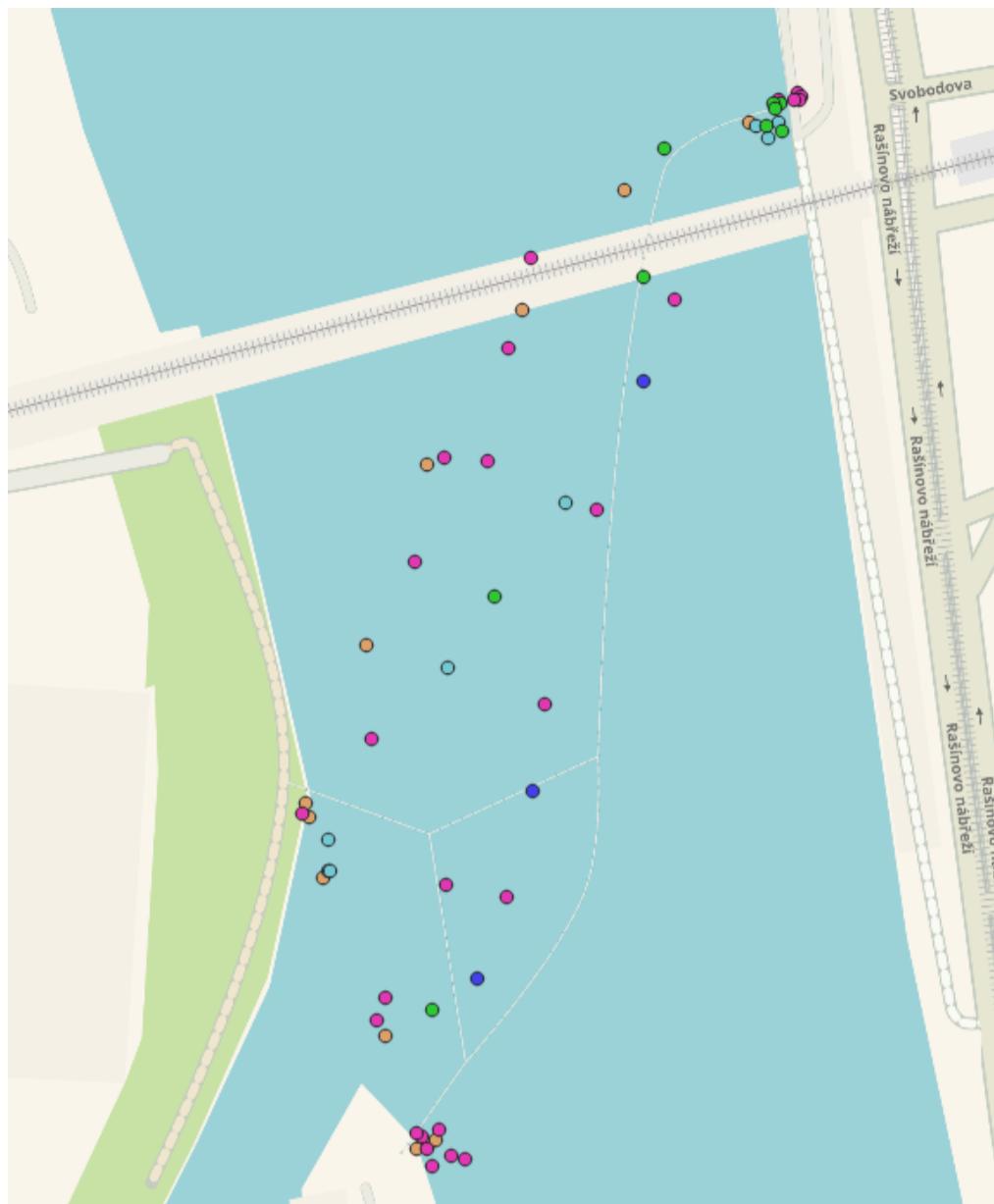
Filtrando en QGIS por `"vehicle_id"="train-9738"`, notar las vías del tren en el mapa



Filtrando en QGIS por `"vehicle_id"='metro-A-2-2'`, notar como por tratarse de un subte ni siquiera se puede seguir claramente su recorrido en el mapa



Si se filtra por `"vehicle_id"='service-0-9250'` se observa que solo se pudo recolectar un punto, lo que impidió su map match



Si se filtra por "vehicle_id"='service-4-308' (cada trip_id corresponde a otro color) se observa que se trata de un ferry que realizó varios viajes, que lógicamente no pudieron ser procesados por Valhalla

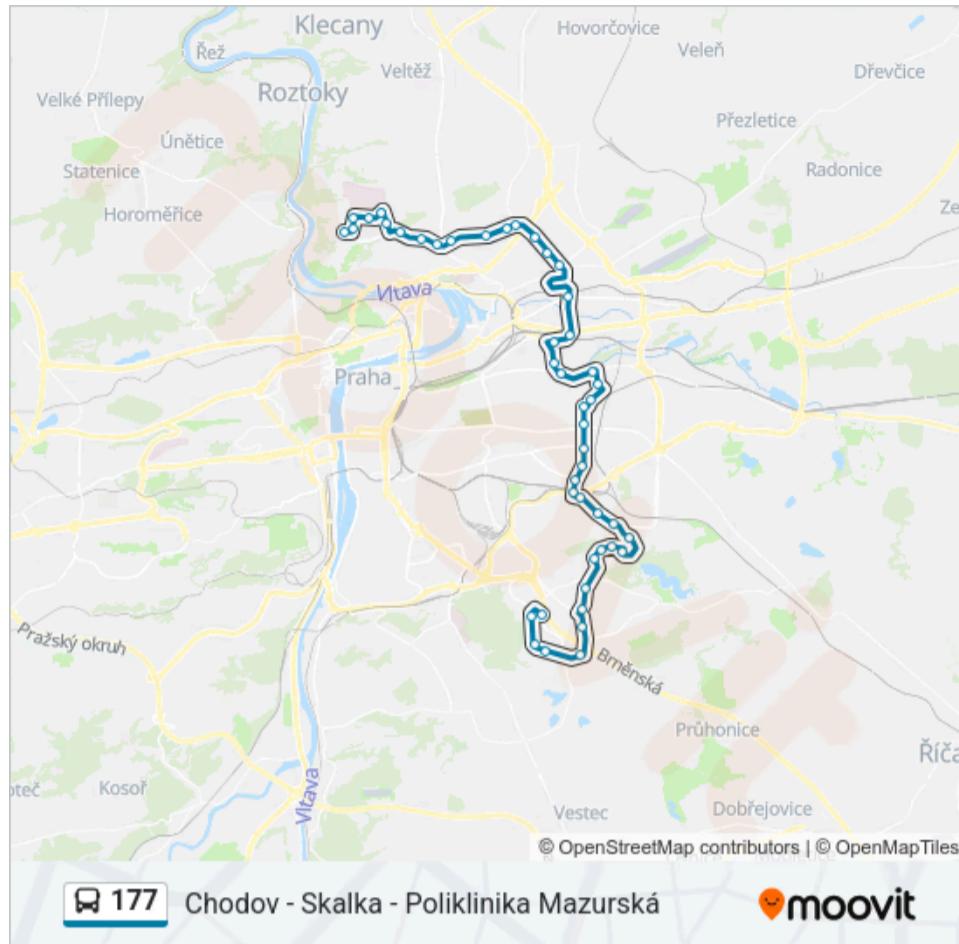
En general, se puede observar que las trayectorias fallidas se debieron o bien a que los datos correspondían a vehículos cuyas rutas no pudieron ser resueltas por Valhalla (lo que provocó que se devolviera `None` en los puntos del map matching), o a trayectorias comunes que fallaron por problemas de GPS o puntos insuficientes.

Importación a MobilityDB

Se utilizó el script para extraer datos el 11/7/2025, entre las 15:26 y 16:25 hora de Argentina (19:26 a 20:25 hora de República Checa). A partir de los datos `map_matched_positions` y `map_matched_shapes`, se generan las tablas `realtime_positions` y `realtime_shapes`. Estos archivos se pueden encontrar en la [carpeta de drive](#).

Se implementó un script `mdb_importer_realtime_new.sql`, que a partir de los puntos de los shapes (obtenidos con `ST_DumpPoints`), les asigna tiempos a través de una interpolación temporal por segmento.

Para ver un ejemplo de interpolación temporal, se muestra a continuación el trip '177_2261_250708' :



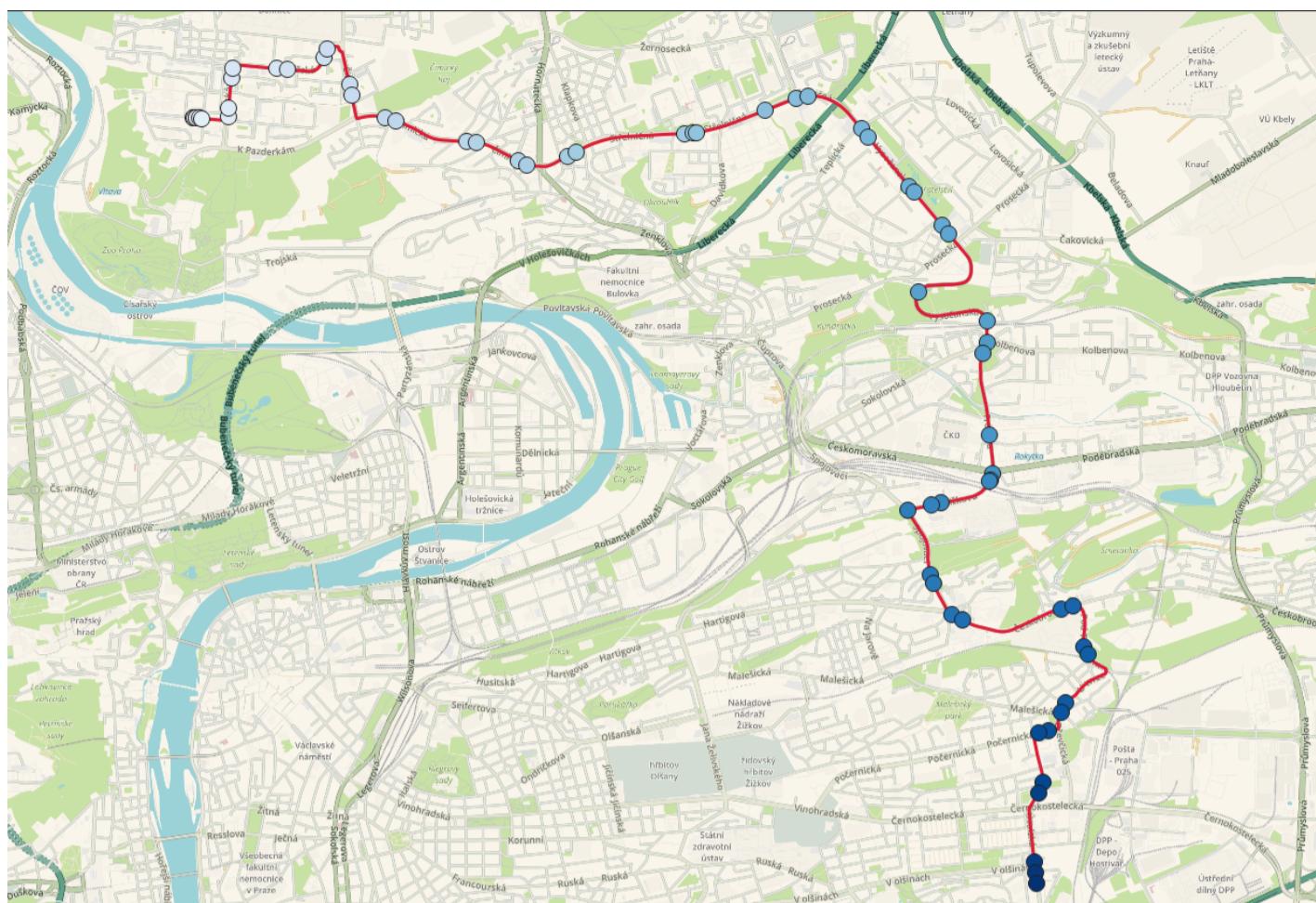
Mapa del trayecto de la línea de colectivo 177 de Praga extraído de https://moovitapp.com/index/en/public_transit_line-177-Prague-1684-775226-371153-0

Si se quiere ver la trayectoria junto con los puntos originales:

```
ALTER TABLE realtime_positions ADD COLUMN geom geometry;
UPDATE realtime_positions SET
geom = ST_SetSRID(ST_MakePoint(Longitude, Latitude), 4326)

SELECT row_number() over () as id, geom, timestamp
FROM realtime_positions
WHERE trip_id = '177_2261_250708'

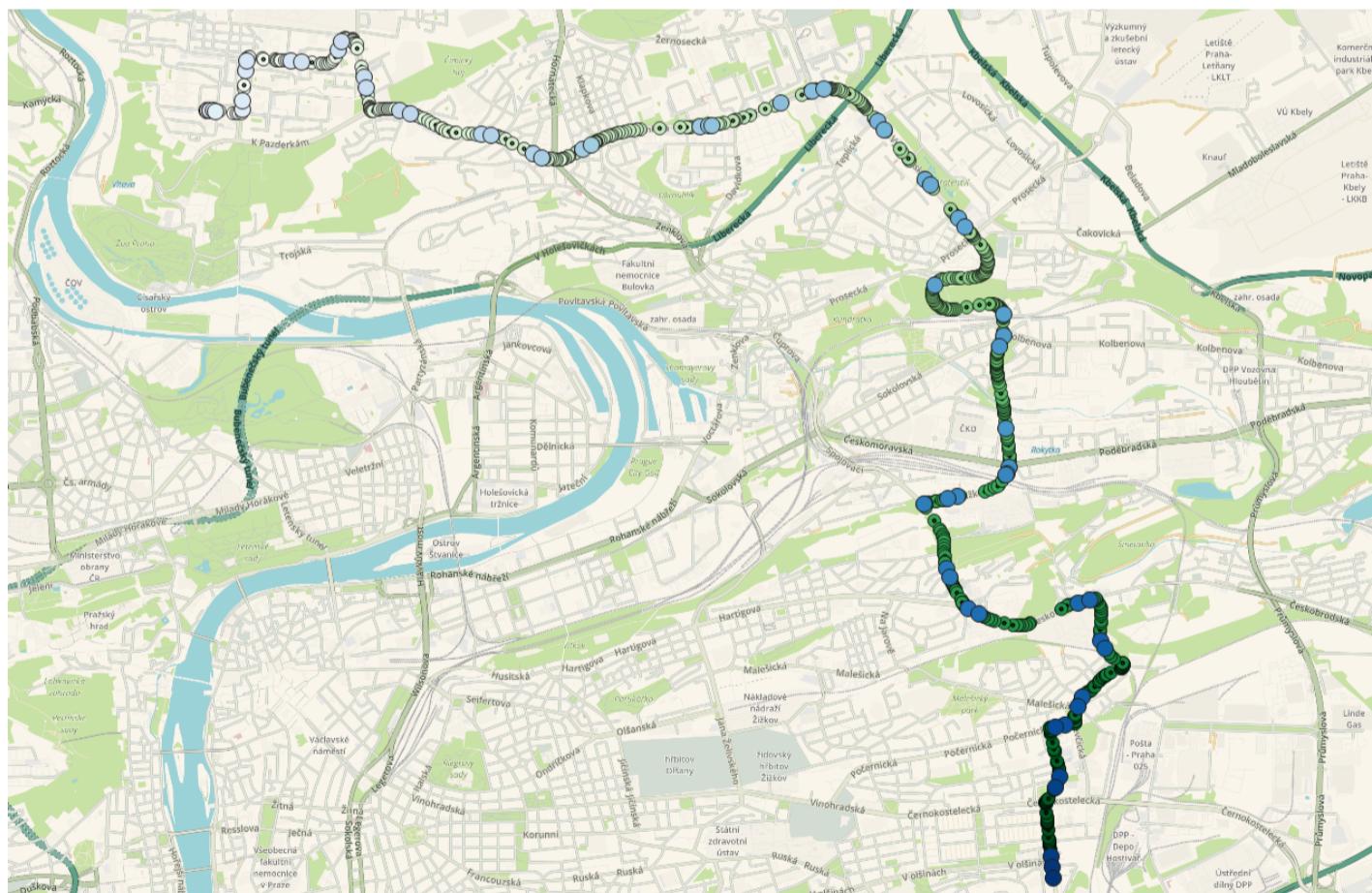
SELECT * FROM realtime_trips_mdb
WHERE trip_id = '177_2261_250708'
```



Geometría de la trayectoria del colectivo 177 luego de la interpolación, junto a los matched points originales (map matcheados). El color de los puntos se va oscureciendo para indicar el paso del tiempo.

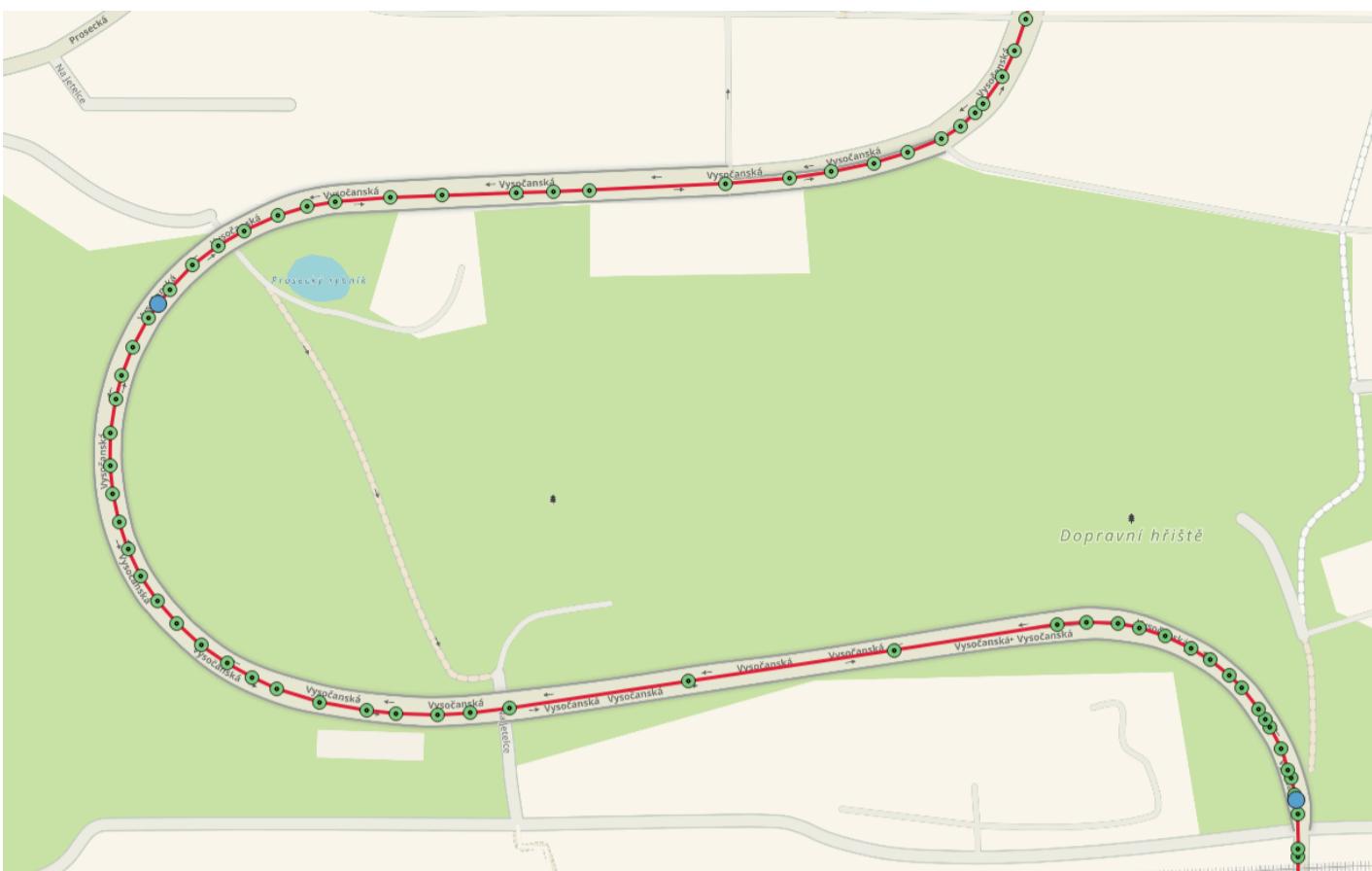
Para visualizar los puntos del shape con tiempo interpolado:

```
SELECT row_number() over () as id, point_geom, interpolated_time FROM interpolated_shape_points
WHERE trip_id = '177_2261_250708'
```



Geometría de la trayectoria del colectivo 177 luego de la interpolación, junto a los matched points originales (map matcheados) y los puntos del shape con timestamp interpolado. Los círculos azules vacíos representan puntos reales y los que tienen un punto en el medio, son puntos con tiempo interpolado. El color de los puntos se va oscureciendo para indicar el paso del tiempo.

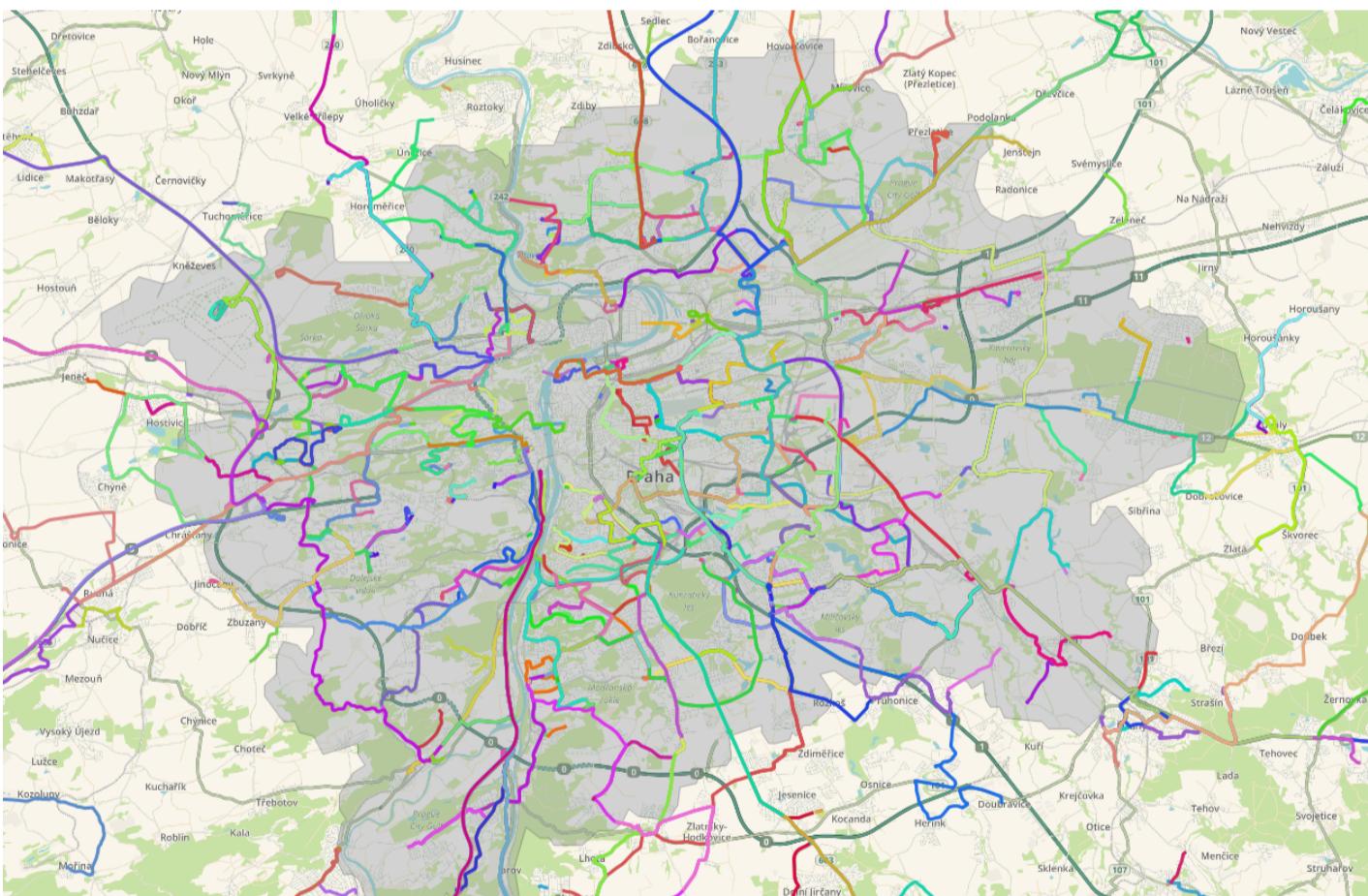
Es interesante notar como Valhalla agrega más puntos a la geometría dentro de curvas y rutas no regulares, y menos en segmentos rectos.



Geometría de la trayectoria del colectivo 177 en una curva. Notar como Valhalla agrega mas puntos para poder representar fielmente las curvas.

Se muestran a continuación los 1063 trips de autobuses que quedaron luego de todo el proceso de filtrado y map matching:

```
SELECT * FROM realtime_trips_mdb t
JOIN routes r ON t.route_id = r.route_short_name AND r.route_type = '3'
```



Visualización de los trips de autobuses luego del proceso de filtrado y map matching

Alucinaciones de Valhalla

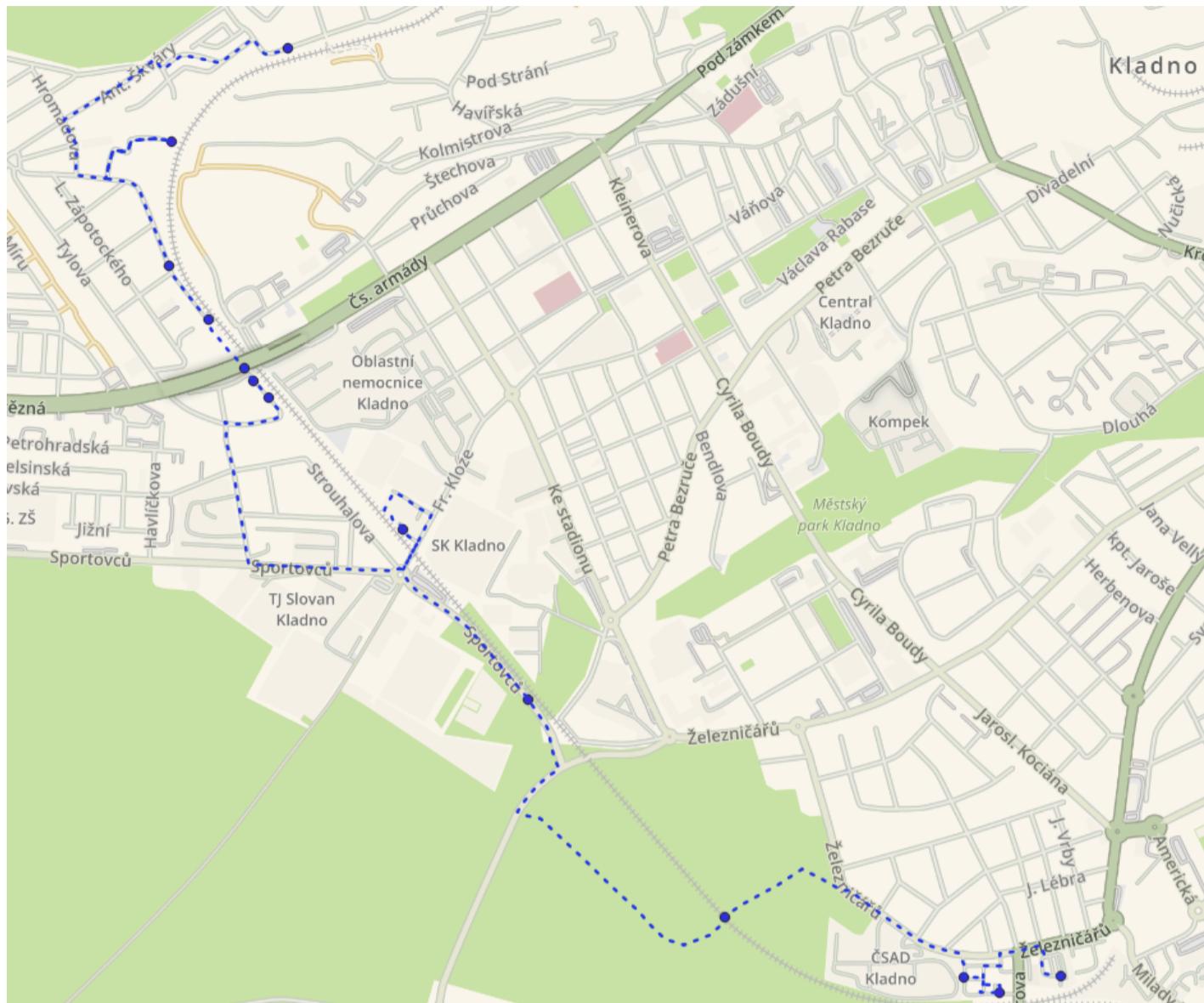
Haciendo un análisis más profundo de las trayectorias, se encontraron casos donde Valhalla tomó caminos incorrectos para el tipo de transporte elegido.

Se muestran en QGIS los puntos el shape del trip [1305_9832_241215](#), que corresponde al tren [train-9832](#).

```
SELECT DISTINCT vehicle_id FROM realtime_positions
WHERE trip_id = '1305_9832_241215';
```

```
SELECT * FROM realtime_positions WHERE trip_id = '1305_9832_241215'
```

```
SELECT * FROM realtime_shapes WHERE trip_id = '1305_9832_241215'
```



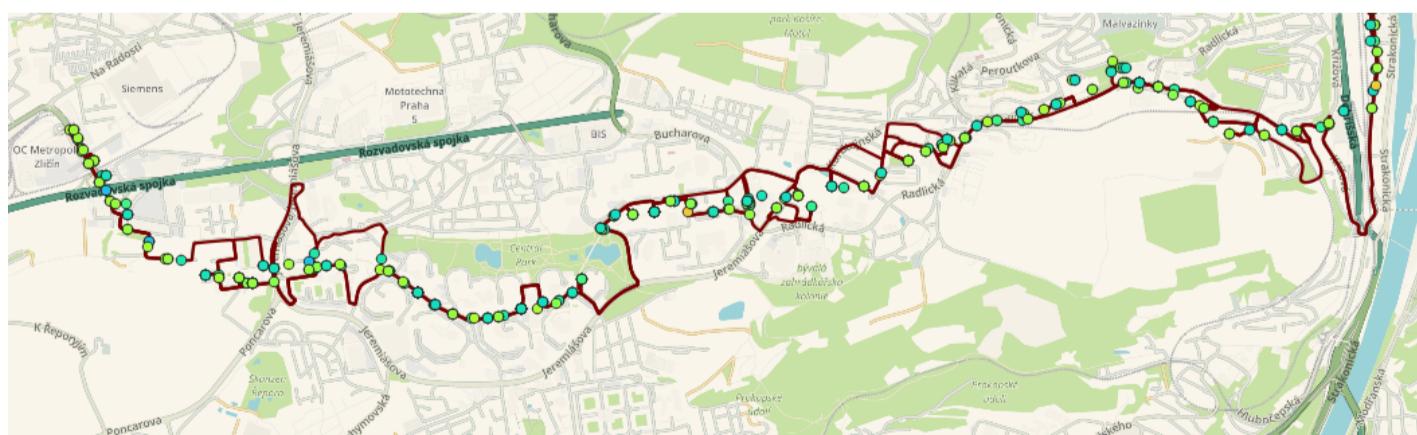
Trip 1305_9832_241215

Lo que se observa en el mapa es que Valhalla no fue capaz de reconocer que se trata de un tren, y en vez de fallar, logró aproximarlo al recorrido que haría un auto. Esto solo se podría solucionar si Valhalla permitiera trabajar con costing models distintos dependiendo el tipo de vehículo, de modo de utilizar la ruta que verdaderamente corresponda a cada tipo de vehículo.

Se encontró que este problema ocurre para todos los trenes y subways.

```
SELECT * FROM realtime_shapes s
JOIN routes r ON s.route_id = r.route_short_name
WHERE route_type = '1'
```

```
SELECT * FROM realtime_positions s
JOIN routes r ON s.route_id = r.route_short_name
WHERE route_type = '1'
```



Recorridos de subtes (cada color representa un trip_id distinto) ajustados para que sean parte del camino de los autos.

Dado que trenes y subtes no tienen una proporción muy significativa sobre el total, no se considera muy problemático mantener sus datos, pero frente a un mayor volumen de datos, sería deseable eliminarlos.

```
SELECT COUNT(*) FROM realtime_shapes s
JOIN routes r ON s.route_id = r.route_short_name
WHERE route_type = '1' OR route_type = '2'; -- 80 sobre 2219 = 3.6% de los datos

SELECT COUNT(*) FROM realtime_shapes s
JOIN routes r ON s.route_id = r.route_short_name
JOIN realtime_trips_mdb t ON t.trip_id = s.trip_id
WHERE route_type = '1' OR route_type = '2'; -- 27 sobre 1338 = 2% de los datos
```

Comparación de los tiempos de demora sobre los trips

Se presenta a continuación una tabla `trip_stops_rt`, como se mostró en el libro de la cátedra, para ver los tiempos de demora sobre cada stop, comparando los datos de gtfs real time con los de scheduled.

```
DROP TABLE IF EXISTS trip_stops_rt;
CREATE TABLE trip_stops_rt AS
WITH Temp AS (
    SELECT
        t.trip_id AS actual_trip_id,
        ad.trip_id AS schedule_trip_id,
        shape_id,
        s.stop_id,
        s.stop_name,
        ad.stop_sequence,
        s.stop_loc::geometry,
        t_arrival AS schedule_time,
        nearestApproachInstant(t.trip, s.stop_loc) AS stop_instant,
        nearestApproachDistance(t.trip, s.stop_loc) as nearest_distance
    FROM
        realtime_trips_mdb t,
        arrivals_departures ad,
        stops s
    WHERE
        ad.date = '2025-07-11'::timestamp
            AND t.startdate::date = '2025-07-11'
            AND ad.stop_id = s.stop_id
            AND t.trip_id=ad.trip_id
            AND nearestApproachDistance(t.trip, s.stop_loc) < 10
)
SELECT
    actual_trip_id,
    schedule_trip_id,
    shape_id,
    stop_id,
    stop_name,
    stop_sequence,
    stop_loc,
    schedule_time,
    getTimestamp(stop_instant) AS actual_time,
    age(getTimestamp(stop_instant), schedule_time),
    nearest_distance,
    ST_Transform(getValue(stop_instant), 4326) AS trip_geom
FROM Temp;
```

Si calculamos la diferencia promedio entre scheduled y real time, llegamos a los siguientes resultados:

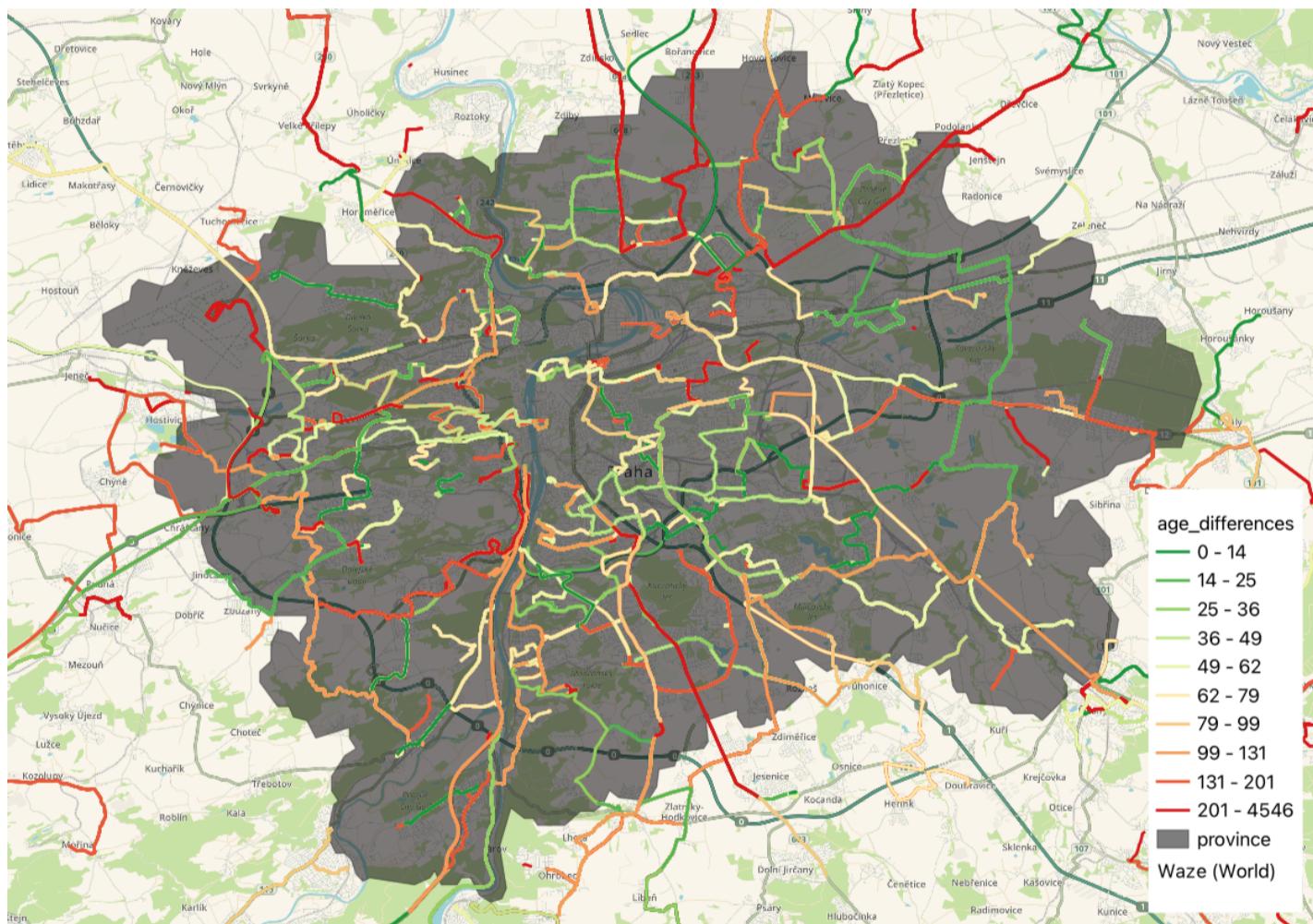
```
SELECT AVG(age) AS average_age, AVG(EXTRACT(EPOCH from age)) AS average_age_seconds FROM trip_stops_rt
```

	average_age interval	average_age_seconds numeric
1	00:01:08.297892	68.2978923374286423

Se observa que es un poco más de un minuto de tiempo extra

Ahora se visualizarán los viajes de autobuses, marcando aquellos que tienen la peor diferencia absoluta con sus tiempos scheduled.

```
SELECT
    r.traj,
    ABS(EXTRACT(EPOCH FROM t.age)) AS age,
    r.trip_id
FROM
    realtime_trips_mdb r
JOIN trip_stops_rt t ON t.actual_trip_ID = r.trip_id
JOIN routes ro ON ro.route_short_name = r.route_id AND ro.route_type = '3'
```



Visualización de los segmentos con la diferencia de tiempos entre los viajes planeados y los reales

Se puede ver que los tiempos son bastante buenos (la gran mayoría tiene menos de 5 minutos de diferencia), aunque claramente hay outliers.

```
SELECT
    COUNT(*)
FROM
    realtime_trips_mdb r
JOIN trip_stops_rt t ON t.actual_trip_ID = r.trip_id
JOIN routes ro ON ro.route_short_name = r.route_id AND ro.route_type = '3'
WHERE
    ABS(EXTRACT(EPOCH FROM t.age)) > 600;
```

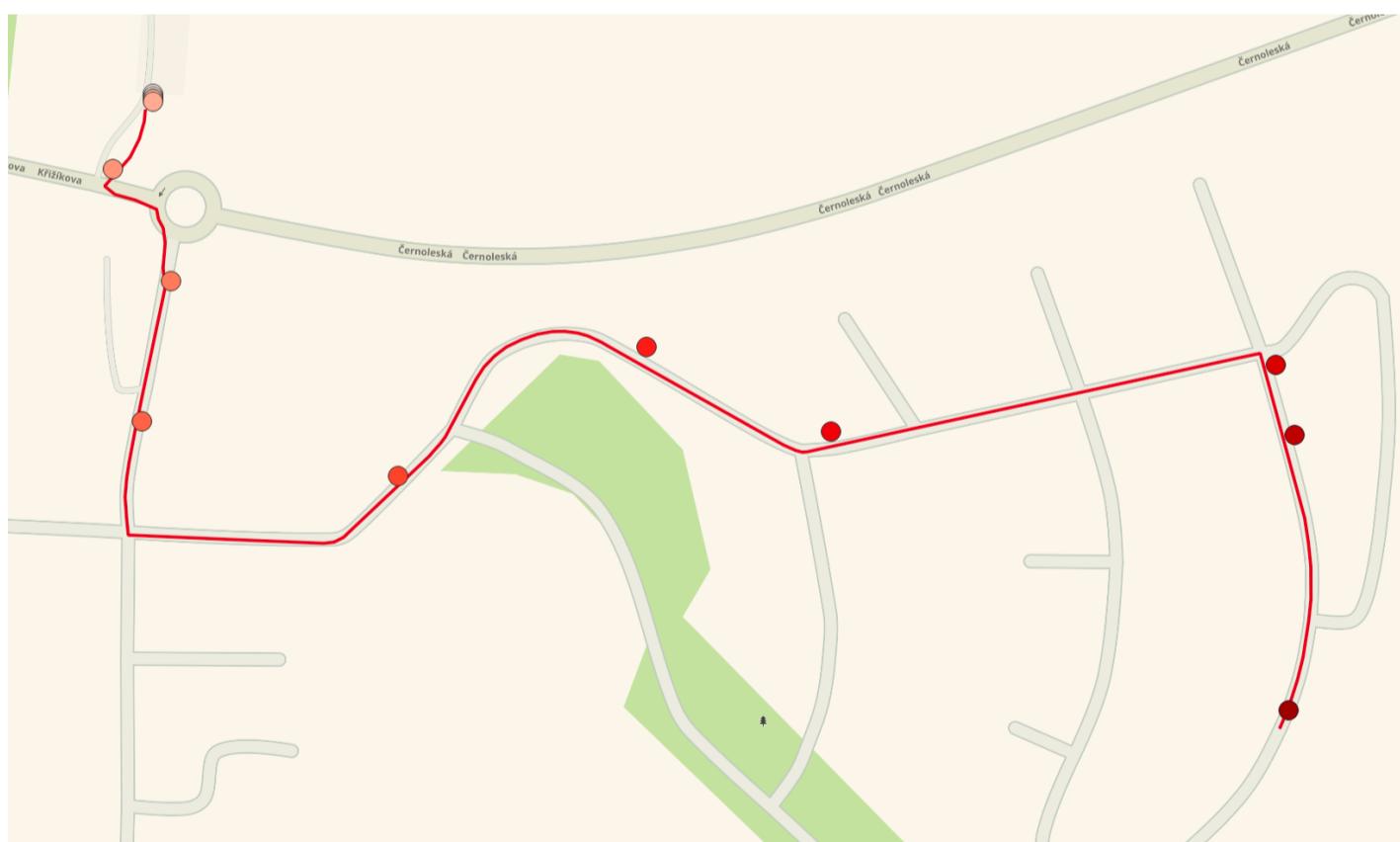
	count	bigint
1	99	

Si hacemos un análisis de cuantos trips tienen mas de 10 minutos de diferencia con el tiempo scheduled, encontramos que hay tan solo 99 de los 1063 trips.

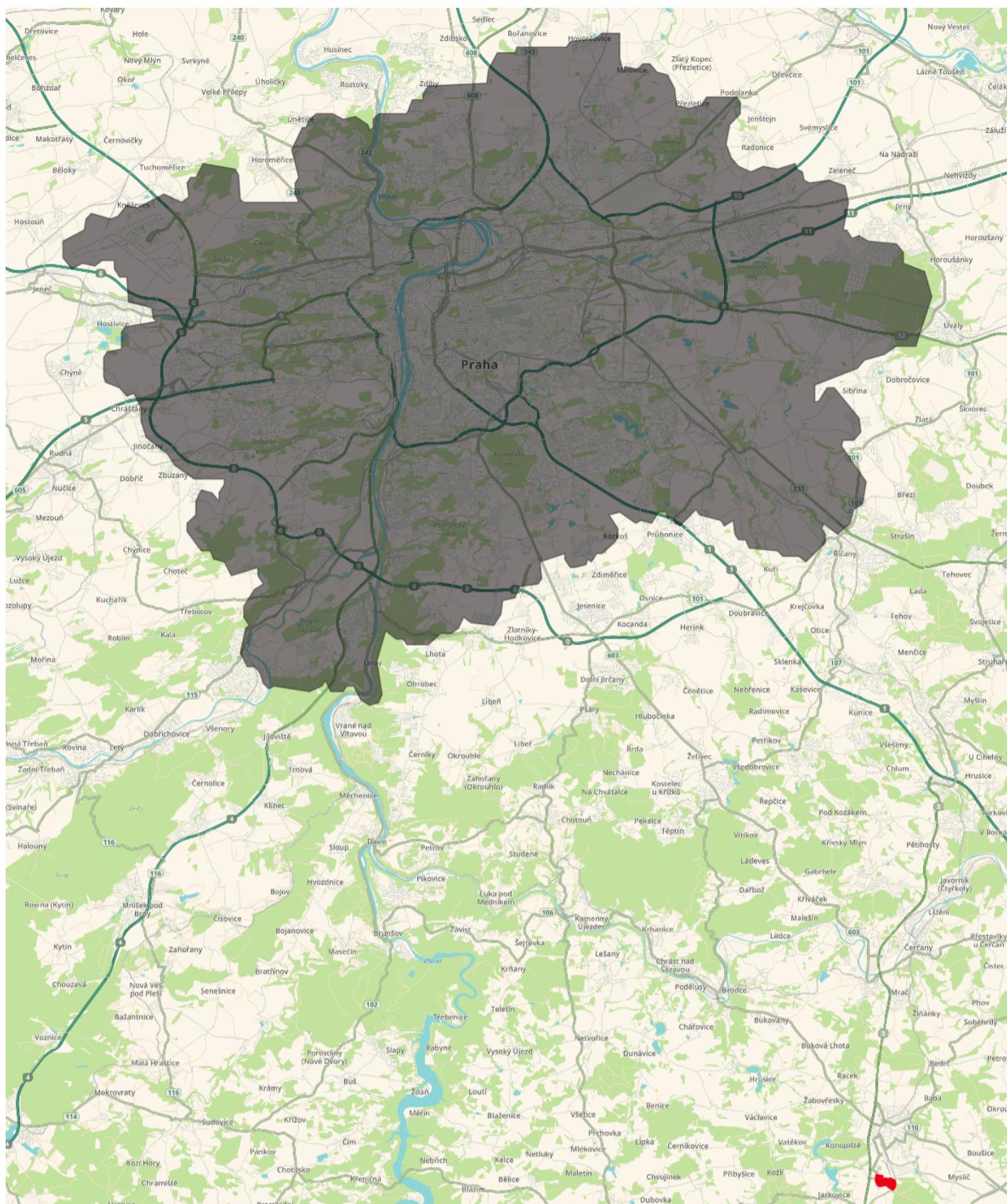
Se obtuvo la ruta que más demora tiene, y se obtuvo que es de la línea 1762. Al investigar, se pudo observar que pasa por Benešov: un pueblo alejado, ubicado a unos 40 kilómetros de Praga.

```
-- Para obtener el trip con más retraso
SELECT
    ABS(EXTRACT(EPOCH FROM t.age)) AS age,
    r.trip_id,
    r.traj
FROM realtime_trips_mdb r
JOIN trip_stops_rt t ON t.actual_trip_ID = r.trip_id
JOIN routes ro ON ro.route_short_name = r.route_id AND ro.route_type = '3'
ORDER BY ABS(EXTRACT(EPOCH FROM t.age)) DESC
LIMIT 1;

-- Para obtener las posiciones de los puntos del trip
SELECT
    ST_Transform(ST_Point(longitude, latitude, 4326), 5514) AS geom,
    ROW_NUMBER() OVER () AS id,
    timestamp
FROM realtime_positions
WHERE trip_id = '1762_803_250626'
ORDER BY timestamp;
```



Recorrido del trip con más retraso y posiciones de los puntos utilizados antes de proyectarlos sobre el trip



Ubicación en el mapa del trip con más retraso

	 <code>st_transform</code> geometry	 <code>id</code> bigint	 <code>timestamp</code> timestamp with time zone
1	0101000208A150000929A130A494026C189B2702D5F7E30C1	1	2025-07-11 18:26:01.537712+00
2	0101000208A1500006E43DB9E494026C157AAB842607E30C1	2	2025-07-11 18:26:22.494662+00
3	0101000208A150000EFA483074A4026C1B5203C3E617E30C1	3	2025-07-11 18:34:27.511447+00
4	0101000208A1500005C8140F4A4026C16DFF735A617E30C1	4	2025-07-11 18:44:39.488002+00
5	0101000208A1500005374DCA34A4026C1BBF6BB6F627E30C1	5	2025-07-11 18:56:36.169303+00
6	0101000208A150000A9B0AD90784026C1368681AC7F7E30C1	6	2025-07-11 19:04:18.822411+00
7	0101000208A150000071E9CB4F4026C165ADE841B77E30C1	7	2025-07-11 19:04:40.034191+00
8	0101000208A1500009FE21EF97C4026C1DD9DFE1AF77E30C1	8	2025-07-11 19:05:01.675525+00
9	0101000208A150000FD27069D933F26C1804536AB207F30C1	9	2025-07-11 19:05:22.507313+00
10	0101000208A150000773507339B3E26C1CB633BB7F37E30C1	10	2025-07-11 19:05:43.487664+00
11	0101000208A15000024E6747AF93D26C1F8A9BDF2267F30C1	11	2025-07-11 19:06:04.449793+00
12	0101000208A15000013BA8566503C26C12F8DA68F237F30C1	12	2025-07-11 19:06:25.467429+00
13	0101000208A1500007736C761473C26C1738E20B7457F30C1	13	2025-07-11 19:06:46.403922+00
14	0101000208A150000C7EAD0B703C26C1105FD599C57F30C1	14	2025-07-11 19:07:07.383947+00

En el resultado de la query se observa que los intervalos de tiempo entre cada una de las posiciones registradas son bastante largos. Esto puede deberse a una baja frecuencia en la toma de datos por parte del sistema de seguimiento o

posibles problemas de cobertura o conectividad en la zona (la lejanía de la ciudad podría influir en la regularidad y calidad del sensado).

Comparación de velocidades

De manera análoga al análisis de tiempos realizado para scheduled, se presentan las velocidades de los segmentos, pero ahora para el día 11 de julio (como en el resto de las consultas real time).

Además, se calcula la diferencia con las velocidades scheduled, para poder realizar la comparación.

Para esto, primero se obtienen los segmentos de manera análoga a lo realizado en el libro.

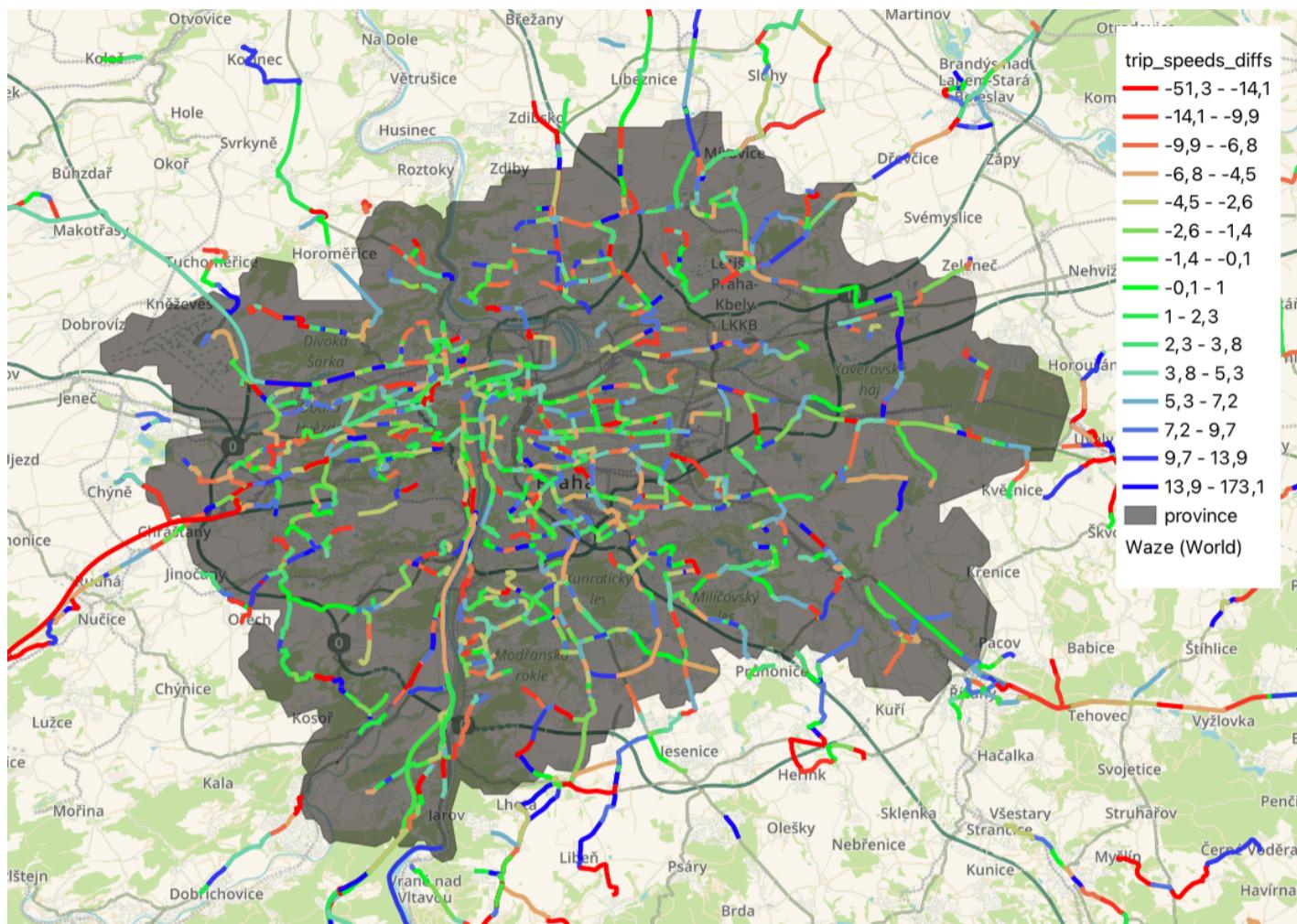
```
DROP TABLE IF EXISTS trip_segments_rt;
CREATE TABLE trip_segments_rt AS
SELECT
    actual_trip_id,
    schedule_trip_id,
    shape_id,
    stop_id AS end_stop_id,
    schedule_time AS end_time_schedule,
    actual_time AS end_time_actual,
    LAG(stop_id) OVER (
        PARTITION BY actual_trip_id
        ORDER BY stop_sequence
    ) AS start_stop_id,
    LAG(schedule_time) OVER (
        PARTITION BY actual_trip_id
        ORDER BY stop_sequence
    ) AS start_time_schedule,
    LAG(actual_time) OVER (
        PARTITION BY actual_trip_id
        ORDER BY stop_sequence
    ) AS start_time_actual
FROM
    trip_stops_rt;
```

```
DROP MATERIALIZED VIEW IF EXISTS trip_speeds_diffs;
CREATE MATERIALIZED VIEW trip_speeds_diffs AS
SELECT
    ROW_NUMBER() OVER () as id,
    t.start_stop_id,
    t.end_stop_id,
    AVG(s.seg_length / EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) * 3.6) AS speed_kmh,
    AVG(s.seg_length / EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) * 3.6) AS speed_kmh_actual,
    AVG(s.seg_length / EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) * 3.6) - AVG(s.seg_length / EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) * 3.6) AS diff,
    s.seg_geom
FROM
    trip_segments_rt t,
    trip_segs s
WHERE
    t.start_stop_id = s.stop1_id
    AND t.end_stop_id = s.stop2_id
    AND t.shape_id = s.shape_id
    AND t.start_time_actual IS NOT NULL
    AND EXTRACT(EPOCH FROM (t.end_time_actual - t.start_time_actual)) > 0
    AND EXTRACT(EPOCH FROM (s.stop2_arrival_time - s.stop1_arrival_time)) > 0
    AND s.seg_geom IS NOT NULL
```

GROUP BY

```
s.seg_geom,  
t.start_stop_id,  
t.end_stop_id;
```

Visualizándolo en QGIS:



Diferencias de velocidades entre realtime y scheduled.

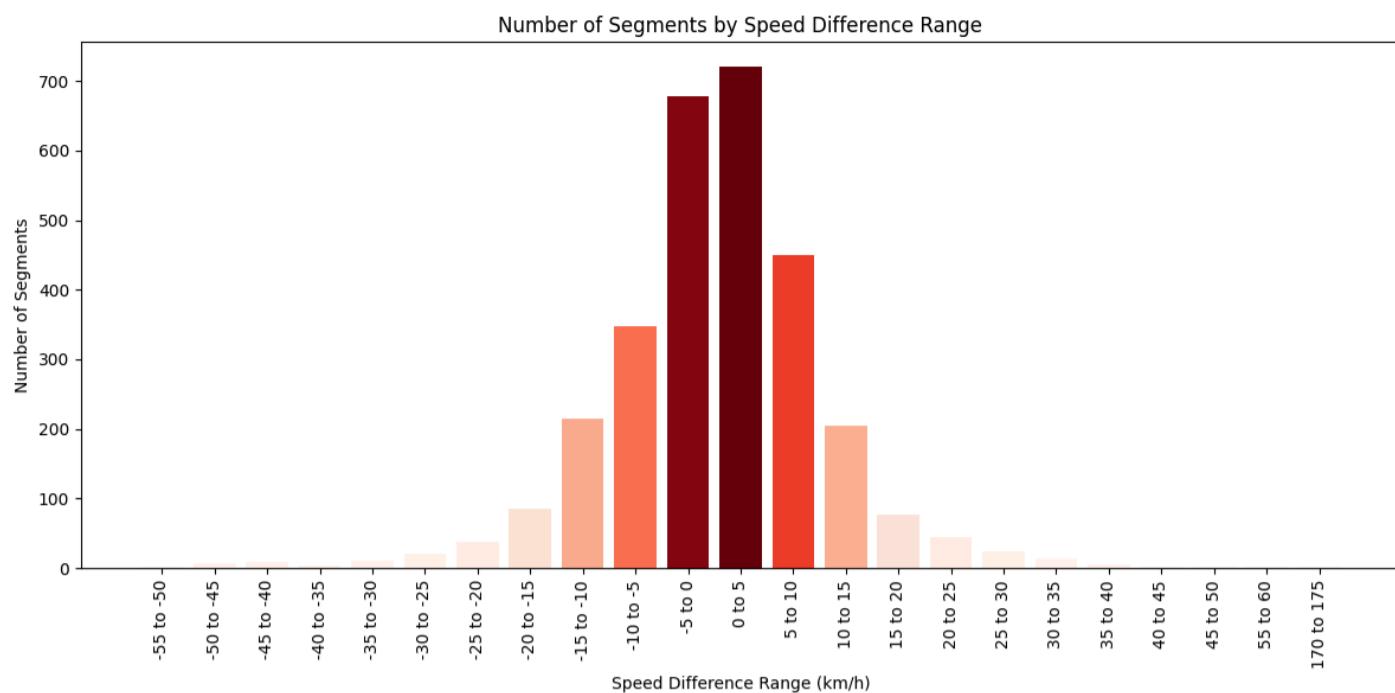
Los colores del mapa facilitan la interpretación de estos desvíos:

- Los segmentos coloreados en azul representan los casos donde la velocidad real fue mayor a la planificada, es decir, los vehículos circularon más rápido de lo esperado.
- Los segmentos en rojo corresponden a situaciones donde la velocidad planificada fue mayor que la real, lo que indica que hubo demoras o los vehículos circularon más lento de lo previsto.

Por ejemplo, se observa, hacia la izquierda en rojo la ruta D5 que sale de Praga. Esto indica que en ese tramo la velocidad real fue mucho menor a la esperada, posiblemente debido a congestión o demoras al salir de la ciudad. Este tipo de situaciones suelen ser comunes en accesos o salidas urbanas, donde el tráfico intenso afecta significativamente la circulación del transporte público.

Se podría reducir la congestión en las salidas de la ciudad dando prioridad al transporte público con carriles exclusivos, gestionando los semáforos para favorecer su paso y restringiendo el acceso de autos particulares en horarios críticos.

Se desarrolló un script en Python ([speed_comparison.py](#)) para analizar la distribución de segmentos de la red de transporte público según sus diferencias de velocidad, agrupándolos en intervalos fijos de 10 km/h.



Histograma de frecuencias de diferencias entre velocidades

Puede observarse que la gran mayoría de los segmentos de la red presentan una diferencia de velocidad entre la real y la planificada dentro del rango de más/menos 10 km/h. Este resultado indica que, en la mayor parte de los casos, la velocidad a la que circulan los vehículos en la realidad se ajusta relativamente bien a lo esperado según los horarios programados.

Aunque el promedio simple de la diferencia es cercano a cero, esto puede ser engañoso porque las diferencias muy positivas y muy negativas tienden a anularse. Por eso, resulta más representativo usar el valor absoluto del promedio, que da aproximadamente 7.5 km/h de diferencia. Además, el 74% de los segmentos presenta una diferencia menor a 10 km/h, confirmando que en la mayoría de los casos la velocidad real es similar a la esperada.

Esto se puede verificar con las siguientes consultas:

```

SELECT AVG(diff) FROM trip_speeds_diffs; -- 0.23 km/h diferencia promedio
SELECT AVG(ABS(diff)) FROM trip_speeds_diffs; -- 7.49 km/h diferencia absoluta promedio

SELECT COUNT(*) FROM trip_speeds_diffs
WHERE ABS(diff) < 10 -- 2195, de los 2959 del total (74.18% aprox)

```