



# Relazione Progetto

Il presente progetto è stato sviluppato da Alberto Pini (212524 10823873) per il corso "PROGETTO DI INGEGNERIA INFORMATICA (5 CFU)" della Prof.ssa M. Fugini.

Nel caso stessi visualizzando questa relazione dal file pdf, per una migliora lettura e formattazione del testo, consiglio di leggere questa relazione sulla pagina qui presente: [Relazione su Notion](#)

## Introduzione e Contesto

- **Obiettivo del Progetto:** Aiutare a sviluppare un'applicazione per l'accessibilità in vista delle Olimpiadi Milano-Cortina 2026
- **Background:** Evoluzione del progetto [MEP](#) (Map for Easy Paths) precedentemente sviluppato al Politecnico di Milano a cura della Prof.ssa S. Comai
- **Supervisori:** Prof.ssa S. Comai e Prof. A. Campi
- **Motivazione:** Necessità di integrare dati provenienti da OSM per la creazione di un programma in grado di trovare un percorso esente da barriere che possono impedire allo specifico utente il raggiungimento della destinazione.

## 1. Estrazione Dati da OSM con Overpass-QL

La prima fase del progetto ha richiesto l'estrazione da OpenStreetMap (OSM) di punti d'interesse che potessero rappresentare barriere o facilitatori per persone con varie disabilità intenzionate a muoversi nell'area di Milano e dintorni. Per questo scopo, è stato necessario apprendere il funzionamento di OSM e del linguaggio di interrogazione Overpass-QL.

### 1.1 OSM e i suoi Elementi

[OSM](#) è un vasto progetto opensource che si occupa di mappare l'intero pianeta. La peculiarità è che rende disponibile al pubblico questo servizio e permette a chiunque di modificare e aggiornare un'unica grande mappa.

Il modello concettuale di OSM è composto da **elementi**. Ci sono 3 tipi di elementi:

- `nodes`: definiscono uno specifico punto
- `ways`: definiscono un polygon (foreste, costruzioni, aree) o un polyline (strade, fiumi)
- `relations`: definiscono come funzionano gli elementi tra di loro

#### Node

consiste in un punto che è definito da: latitudine, longitudine e un identificativo

- `id`: undefined 64-bit integer  $\geq 1$ 
  - l'id è unico tra i nodi  $\Rightarrow$  potrebbe esserci una way o una relation con lo stesso id
  - id negativi sono usati per denotare nodi non ancora salvati nel server
- `lat`:  $-90 \leq \text{lat} \leq 90$  con 7 cifre decimali
- `lon`:  $-180 \leq \text{lon} \leq 180$  con 7 cifre decimali

esempio:

```
<node id="25496583" lat="51.5173639" lon="-0.140043" version="1"
      changeset="203496" user="80n" uid="1238" visible="true"
      timestamp="2007-01-28T11:40:26Z">
  <tag k="highway" v="traffic_signals"/>
</node>
```

Normalmente i nodi sono usati per:

- definire strutture o features presenti in un certo posto. In questo caso hanno almeno un tag che determina cosa rappresenta quel nodo
- definire la struttura di una o più ways: possono non avere alcun tag e/o essere parte di una o più ways oppure anche sovrapporsi a livelli differenti

## Way

Soltanamente rappresenta una caratteristica lineare come le strade, muri, fiumi. Una way è definita come una lista ordinata di nodi. Inoltre:

- Presenta almeno un tag oppure partecipa in una relazione.
- Può avere da 2 a 2000 nodi
- Possono essere aperte (zone - aree) o chiuse (strade - fiumi)

Ci sono 3 tipi di way:

- **Open way - open polyline:**
  - primo e ultimo nodo non sono identici (non coincidono)
  - è sempre presente una direzione dettata dall'ordine dei nodi (anche per le cose che non hanno direzione (tipo un muro) o che sono bidirezionali (la maggior parte delle strade o sentieri))
- **Closed way - closed polyline:**
  - l'ultimo nodo è identico al primo
  - si può interpretare sia come una closed way che un'area:
    - una closed way potrebbe rappresentare una recinzione di una proprietà o i limiti di una rotonda stradale
    - se però ha il tag `area=yes` allora dev'essere sempre interpretata come un'area
- **Area - Polygon:**
  - si fa riferimento a tutta l'area interna circondata da una closed way

Esempio di way rappresentante la strada "Clipstone Street":

```
<way id="5090250" visible="true" timestamp="2009-01-19T19:07:25Z" version="8"
    changeset="816806" user="Blumpsy" uid="64226">
    <nd ref="822403"/>
    <nd ref="21533912"/>
    <nd ref="821601"/>
    <nd ref="21533910"/> tutti riferimenti a nodi già definiti
    <nd ref="135791608"/>
    <nd ref="333725784"/>
    <nd ref="333725781"/>
    <nd ref="333725774"/>
    <nd ref="333725776"/>
    <nd ref="823771"/>
    <tag k="highway" v="residential"/>
    <tag k="name" v="Clipstone Street"/>
    <tag k="oneway" v="yes"/>
</way>
```

## Relation

Le **relazioni** sono collezioni di strutture di elementi (nodes, ways e altre relations).

Ogni elemento relation deve avere almeno un tag `type=*` e un gruppo di membri, cioè una lista ordinata di uno o più elementi (nodi, ways e/o altre relazioni).

La funzione delle relazioni è quella di definire connessioni logiche o geografiche tra oggetti diversi. Per esempio tutte le strade che formano la linea dei bus o un lago con la sua isola. Sarebbe però inappropriato usare relazioni

che contengono membri poco associati e distribuiti su un largo territorio (tutte le autostrade d'Italia) (sarebbe una categoria)

Opzionalmente un membro della relazione può avere un `role` che descrive la funzione che ha nella relazione. Per esempio la relazione che descrive un fiume che ha un tributario potrebbe contenere 2 ways con 2 role diversi:

`main_stream` e `side_stream`.

Tecnicamente il limite di membri di una relazione è di 32.000 ma il consiglio è di usarne meno di 300 per relazione. Se servono più di 300 membri allora è bene suddividere in sotto-relazioni e collegare tutto in una super-relazione contenente questi sotto-membri.

```
<relation id="13092746" visible="true" version="7" changeset="118825758"
    timestamp="2022-03-23T15:05:48Z" user="" uid=""/>
<member type="node" ref="5690770815" role="stop"/>
<member type="node" ref="5751940550" role="stop"/>
...
<member type="node" ref="1764649495" role="stop"/>
<member type="way" ref="96562914" role=""/>
...
<member type="way" ref="928474550" role=""/>
<tag k="from" v="Encre"/>
<tag k="name" v="9-Montagnes de Guyane"/>
<tag k="network" v="Agglo'bus"/>
<tag k="not:network:wikidata" v="Q3537943"/>
<tag k="operator" v="CACL"/>
<tag k="ref" v="9"/>
<tag k="route" v="bus"/>
<tag k="source" v="https://www....pdf"/>
<tag k="to" v="Lycée Balata"/>
<tag k="type" v="route"/>
<tag k="website" v="https://www.../"/>
</relation>
```

Una volta specificato il tipo di struttura devo specificare cosa sta descrivendo nel mondo reale. A questo scopo nascono i **tag**

## Tag

I **tag** sono dei metadati che descrivono nodes, ways e relations. Sono proprio il cuore delle query Overpass-Turbo di cui parlerò nel prossimo capitolo.

Strutturalmente non sono altro che un'entry di tipo `key:value`:

- la chiave `key` può descrivere una categoria o una caratteristica e possono essere qualificate tramite prefissi, infissi o suffissi: `prefisso:key:suffisso` per creare super o sub categorie
- il valore `value` che è sempre associato ad una chiave e descrive il dato che la chiave propone. I valori sono sempre obbligatori dopo la chiave!! (`motorcycle:rental=yes`)

## 1.2 Overpass-Turbo QL API

Per estrarre nodi, ways e relations dalla mappa di OSM ho dovuto usare questo tool per l'interrogazione dei DB OSM. Si tratta di un'API a cui si invia come payload della richiesta una query in uno specifico linguaggio creato appositamente per OSM detto **Overpass QL** ([demo](#)), la risposta sarà invece un file JSON contenente gli elementi estratti in un formato simile a questo:

```
{
  "type": "node",
  "id": 281819170,
  "lat": 45.4950751,
  "lon": 9.1970870,
  "tags": {
    "amenity": "cafe"
  }
}
```

```
{
  "type": "node",
  "id": 282314361,
  "lat": 45.4882021,
  "lon": 9.1953454,
  "tags": {
    "addr:city": "Milano",
    "addr:housenumber": "5",
  }
}
```

```

},
{
  "type": "node",
  "id": 282314360,
  "lat": 45.4887935,
  "lon": 9.1943986,
  "tags": {
    "addr:city": "Milano",
    "addr:postcode": "20124",
    "addr:street": "Via Ippo...",
    "amenity": "cafe",
    "check_date": "2024-10-10",
    "name": "Bar Nilo",
    "website": "http://www./",
    "wheelchair": "limited"
  }
},
{
  "addr:postcode": "20124",
  "addr:street": "Viale Fi",
  "amenity": "cafe",
  "brand": "Starbucks",
  "brand:wikidata": "Q37158",
  "check_date": "2024-10-10",
  "cuisine": "coffee_shop",
  "indoor_seating": "yes",
  "name": "Starbucks",
  "note:it": "l'orario sulla p",
  "official_name": "Starbucks",
  "opening_hours": "Mo-Fr 07",
  "outdoor_seating": "yes",
  "takeaway": "yes",
  "wheelchair": "no"
}
},

```

Questi sono 3 esempi di nodi che hanno soddisfatto la ricerca di `"amenity"="cafe"` nell'area metropolitana di milano. In 1.4 ci saranno tanti esempi di questo specifico tipo di query.

## 1.3 Prime due Query

Le primissime query che ho fatto erano per trovare scale con corrimano (come facilitatori per alcune persone con problemi di mobilità) e i marciapiedi inadatti per persone in sedia a rotelle (fondamentalmente quelli più stretti di 0.6m o con fondi non regolari di materiali naturali come la terra o la ghiaia)

I risultati di queste query non sono però stati molto soddisfacenti, almeno non da soli.

Fino a questo punto mi sono infatti focalizzato sul creare delle query il più specifiche possibili, portando quindi tutta la logica in overpass-QL e togliendola dalla futura applicazione che, secondo la mia idea, non avrebbe dovuto far altro che visionare i risultati delle query che quindi sarebbero già stati intrinsecamente classificati.

Ed è proprio qui che mi sono accorto del problema che avrebbe sollevato ed ho quindi deciso di cambiare approccio: Fare query generali che raccolgano il maggior numero possibile di elementi OSM, successivamente ci sarà una fase di elaborazione che riuscirà a fare molto di più con i dati che, con l'approccio precedente, avrei filtrato e mai utilizzato.

## 1.4 Query Generali e JSON

Quindi ricapitolando: al posto di fare poche query molto specifiche per individuare poche cose su dati potenzialmente incompleti è meglio estrarre una maggiore quantità di dati "raw" per poi farli processare maggiormente all'applicazione/programmi che li usano.

Per fare queste query si possono considerare tutti quei tag che potrebbero direttamente o indirettamente servire per l'accessibilità. Per esempio...

### Wheelchair

Tag piuttosto intuitivo che ci rivela se un elemento è relazionato in qualche modo con le sedie a rotelle (una rampa, un ascensore ecc...). Gli elementi ritornati da questa query sono potenziali duplicati degli elementi delle query successive.

```

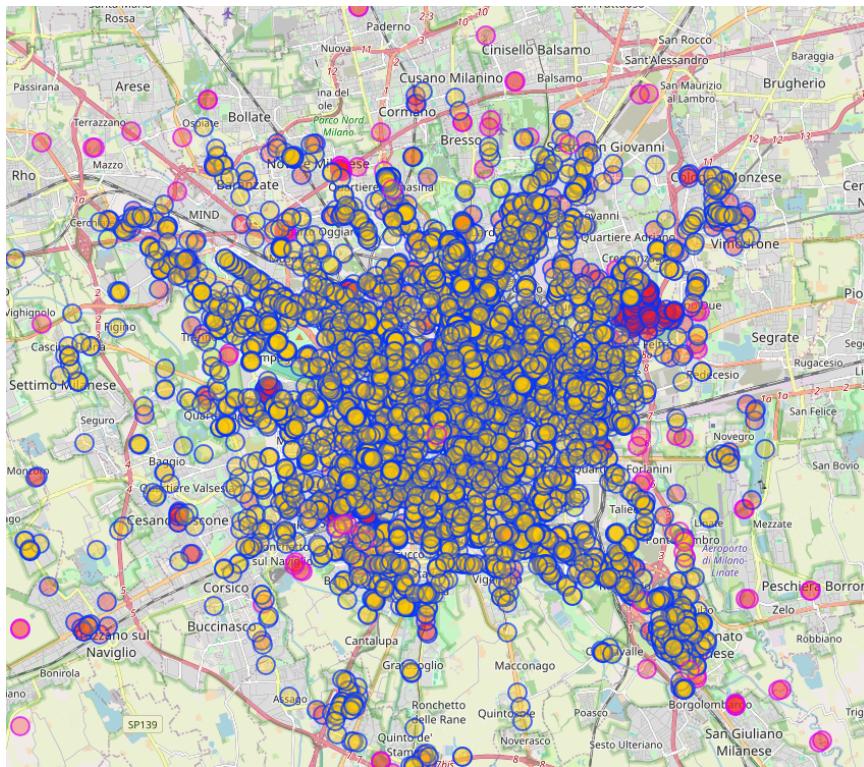
[out:json];

relation(44915) → .milano; // esempio su Milano
.milano map_to_area → .areaMilano;

(
  nwr(area.areaMilano)["wheelchair" ~ "yes|designated"];
  nwr(around.milano:1500)["wheelchair" ~ "yes|designated"];
) → .risultato;

```

.risultato out geom;



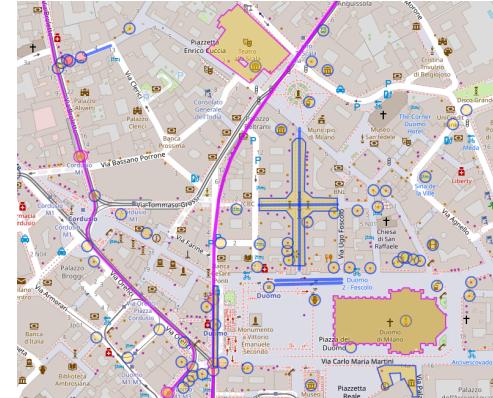
```
"tags": {  
    "nodes": "5381",  
    "ways": "1103",  
    "relations": "125",  
    "areas": "0",  
    "total": "6609"  
}
```

ci sono quindi un totale di 6609 elementi che hanno il tag `wheelchair` pari a `yes` o `designated` (cioè fatto apposta)

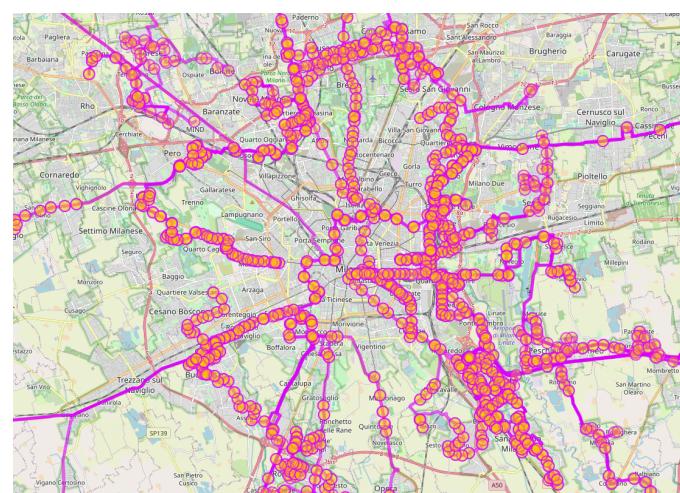
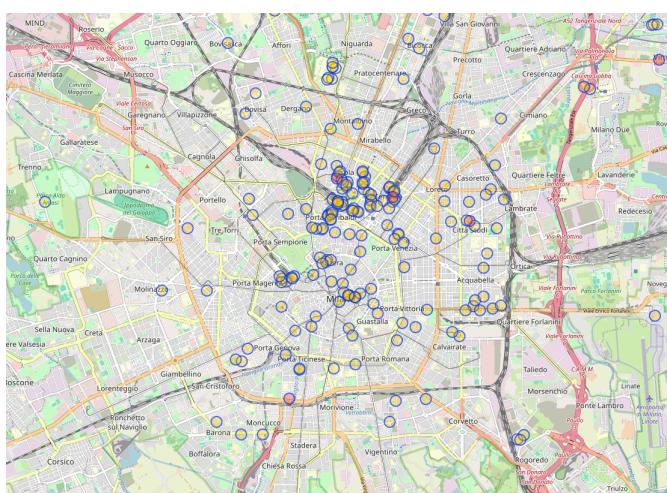
Come si vede dall'immagine qui affianco...

La varietà di tipologie di elementi che ritorna questa query è altissima, così come i dati che risultano dalla query stessa (10 MB).

Quindi sarebbe utile separare tutti questi elementi a seconda della natura della caratteristica mappata (cioè in base a cosa sono).



Per esempio da tutti questi tag posso estrarre i ristoranti/caffè `"amenity"~"restaurant|cafe"` con `wheelchair=yes`. Oppure tutte le linee dei bus `route=bus` con accesso esplicito alla sedia a rotelle ...



Questa metodologia si può applicare a molte altre cose

## Bench

Potrebbe essere utile ricercare tutte le chiavi `amenity` (servizi) di valore `bench` (panchina) nel caso si volesse riposare durante una passeggiata.

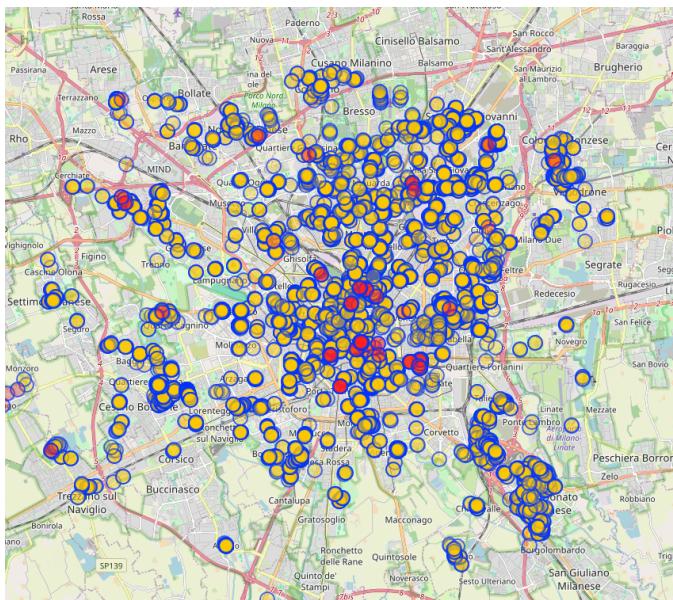
```
[out:json];
```

```
relation(44915) → .milano;  
.milano map_to_area → .areaMilano;  
(  
    nwr(area.areaMilano)[“amenity” = “bench”];
```

```
nwr(around.milano:1500)["amenity" = "bench"];
```

```
) → .risultato;
```

```
.risultato out geom;
```



```
"tags": {  
    "nodes": "6093",  
    "ways": "112",  
    "relations": "0",  
    "areas": "0",  
    "total": "6205"  
}
```

Alcune bench sono delle ways dato che sono lunghe e/o hanno forme strane.

## Elevators

Per trovare gli ascensori si può usare `highway=elevator`:

```
[out:json];
```

```
relation(44915) → .milano;
```

```
.milano map_to_area → .areaMilano;
```

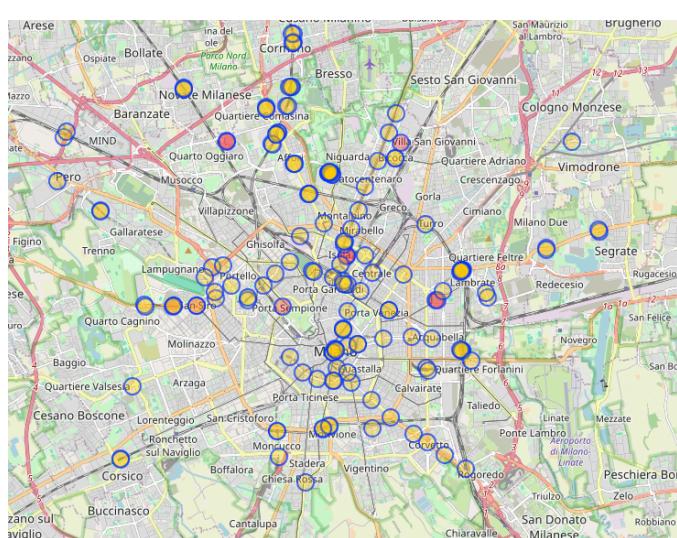
```
(
```

```
    nwr(area.areaMilano)["highway" = "elevator"];  
    nwr(around.milano:1500)["highway" = "elevator"];  
) → .risultato;
```

```
( // togli ascensori non adatti alle sedie a rotelle
```

```
    .risultato; - nwr.risultato["wheelchair" ~ "no"];  
) → .risultato;
```

```
.risultato out center;
```



```
"tags": {  
    "nodes": "160",  
    "ways": "10",  
    "relations": "0",  
    "areas": "0",  
    "total": "170"  
}
```

Alcuni di questi ascensori sono privati o all'interno di edifici ai quali normalmente non è possibile l'accesso.

Inoltre i motivi per cui etichettare gli ascensori come ways sono principalmente 3:

- L'ascensore è un "inclinator" quindi una sorta di funicolare tipo quelle sulle gambe più basse della Tour Eiffel
- Si usa una way per descrivere l'area (dall'alto) che l'ascensore occupa

- Oppure ancora per definire una way verticale composta da nodi sovrapposti ciascuno a livelli diversi per indicare a che piani si ferma l'ascensore

Ascensori/montascale della metro presenti e in servizio (live update) sono visibili da [questo](#) sito dell'ATM. Avrebbe senso fare un web scraper?

## Tactile

Questo è un tag che viene usato quando sul territorio è presente una qualsiasi forma di pavimentazione tattile

[tactile\\_paving](#)

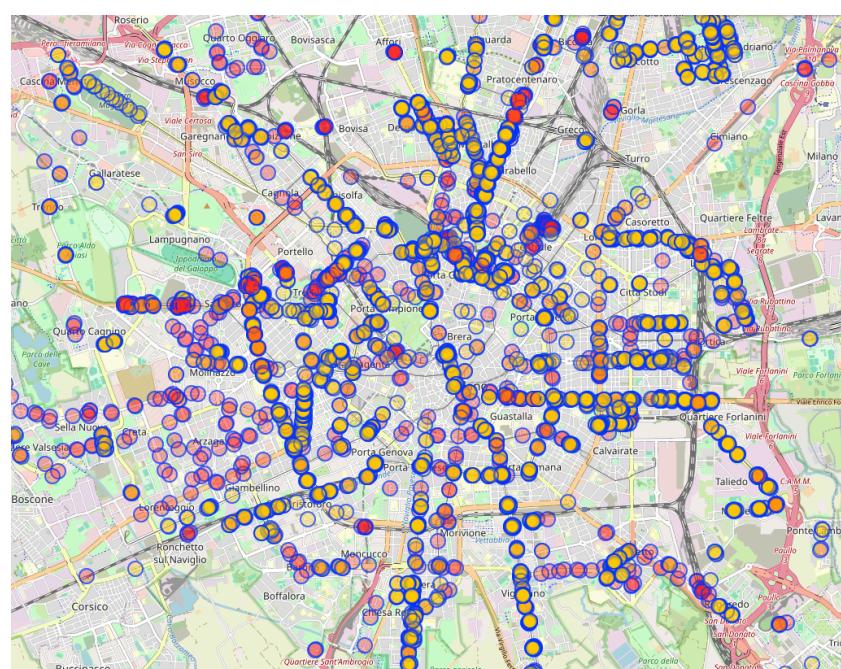
```
[out:json];

relation(44915) → .milano;
.milano map_to_area → .areaMilano;

(
  nwr(area.areaMilano)[tactile_paving=yes];
  nwr(around.milano:1500)[tactile_paving=yes];
) → .risultato;

.risultato out geom;
```

Principalmente si tratta di stazioni di treni, tram e bus ma anche attraversamenti pedonali.



```
"tags": {
  "nodes": "3172",
  "ways": "1163",
  "relations": "0",
  "areas": "0",
  "total": "4335"
}
```

## Sound/Vibration Signal Crossing

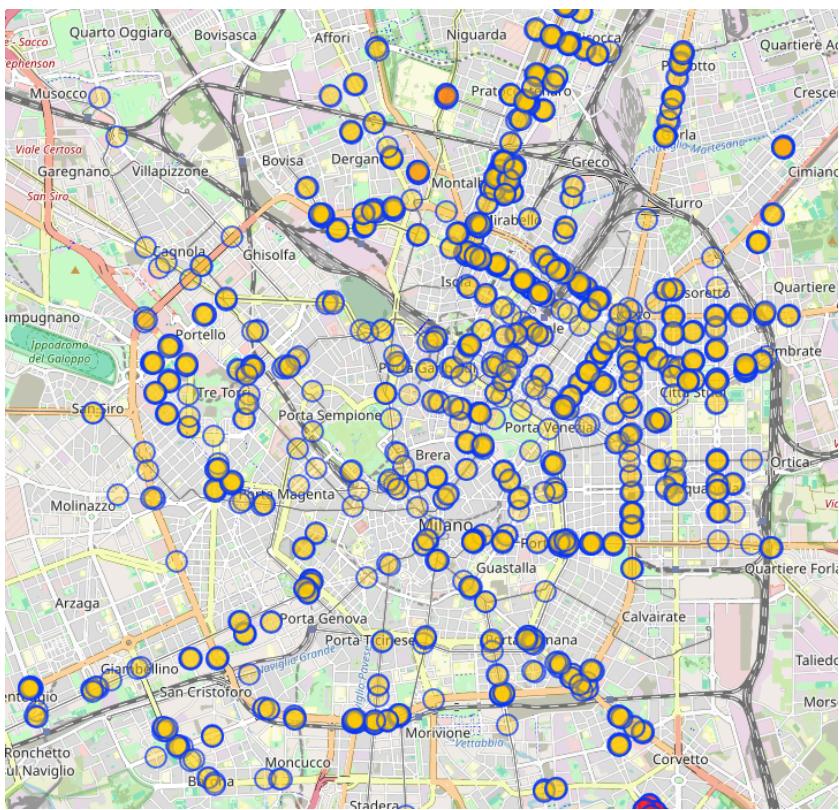
Di utilità diretta sono sicuramente gli attraversamenti pedonali con una qualsiasi tipo di segnale per i non vedenti. [Qui](#) un po' di documentazione sui crossings che sono un po' complicati da mappare.

```
[out:json][timeout:60];

relation(44915) → .milano;
.milano map_to_area → .areaMilano;

(
  nwr(area.areaMilano)
    [~"footway|highway"~"crossing"] [~"traffic_signals:(sound|vibration)"~"yes"];
  nwr(around.milano:1500)
    [~"footway|highway"~"crossing"] [~"traffic_signals:(sound|vibration)"~"yes"];
) → .attraversamentiPedonaliConSegnaleAcustico;

.attraversamentiPedonaliConSegnaleAcustico out center;
```



```
"tags": {
  "nodes": "1228",
  "ways": "29",
  "relations": "0",
  "areas": "0",
  "total": "1257"
}
```

I nodi sono sempre piazzati al centro degli attraversamenti

Si potrebbe estrarre la way dall'attraversamento dato che i nodi trovati dalla query rappresentano solo la posizione dell'intersezione tra una strada e un marciapiede.

## Bagni Pubblici Accessibili o Meno

```
[out:json];

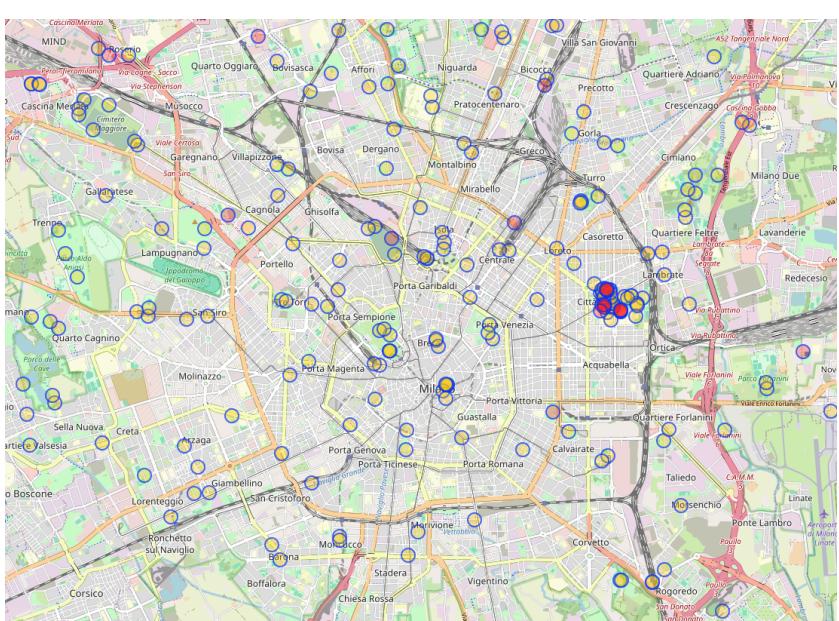
relation(44915) → .milano;
.milano map_to_area → .areaMilano;

(
  nwr(area.areaMilano)["amenity"="toilets"];
  nwr(around.milano:1500)["amenity"="toilets"];
) → .bagniPubblici;

(
  nwr(area.areaMilano)["amenity"="toilets"]["wheelchair" ~ "yes|designated"];
  nwr(around.milano:1500)["amenity"="toilets"]["wheelchair" ~ "yes|designated"];
) → .bagniPubbliciPerSedieARotelle;

(
  nwr(area.areaMilano)["amenity"="toilets"]["wheelchair"="no"];
  nwr(around.milano:1500)["amenity"="toilets"]["wheelchair"="no"];
) → .bagniPubbliciNonAdatti;

.bagniPubblici out geom; ...
```



```
"tags": {
  "nodes": "235",
  "ways": "31",
  "relations": "0",
  "areas": "0",
  "total": "266"
}
```

Poi si potranno fare le varie differenze tra gli insiemi di nodi e ways (0 relations) per trovare ciò che si vuole.

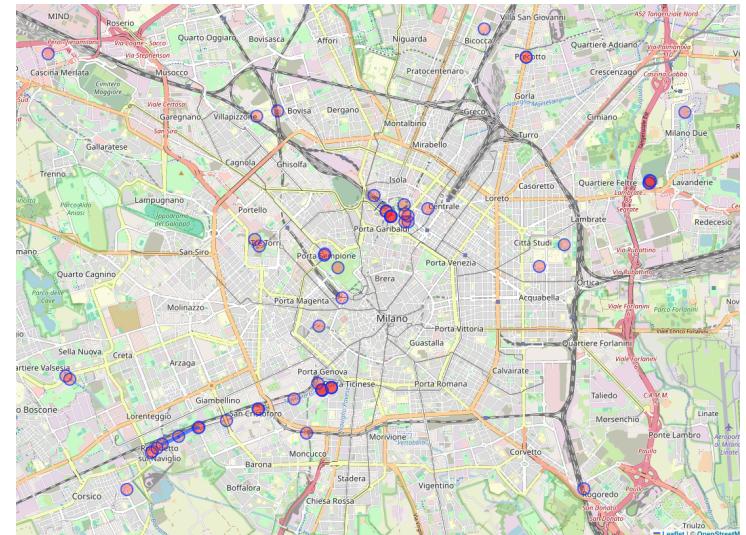
## Ramp / Incline

Conoscere il dislivello di una certa strada potrebbe essere un ostacolo all'accessibilità. `ramp` è un tag spesso usato con `incline` per descrivere dislivelli su elementi di tipo way.

```
( // trova steep o dal 5% in su  
nwr(area.searchArea)["incline"~"^5-9%|[1-9][0-9%|[1-9][0-9][0-9]+%"];  
nwr(area.searchArea)["incline"~"steep"];  
) → .dislivellilmpagnativi;
```

Ho provato questa query su Bormio e Livigno (città host per alcuni dei giochi invernali) ma i risultati sono stati abbastanza scarsi (i centri paese sono generalmente in piano).

```
[out:json];  
  
relation(44915) → .milano;  
.milano map_to_area → .areaMilano;  
  
(  
nwr(area.areaMilano)["ramp"~"yes"];  
nwr(around.milano:1500)["ramp"~"yes"];  
) → .risultato;  
  
.risultato out geom;
```



Risultati un po' scarsi

Per capire il dislivello si può anche usare API come [Google Elevation API](#) o [open-Elevation](#) facendo delle query strategiche per trovare la posizione a inizio e fine (o anche in punti intermedi come i nodi che formano la way) del tratto in questione per poi calcolarne il dislivello.

## Kerb

Il "kerb" sarebbe il lato del marciapiede che può essere rialzato o meno.

Il tag in se `kerb` è usato in più modi (vedi [qui](#)):

- si usa come valore di `barrier=kerb` nelle ways per indicare il percorso che fa (in lunghezza)
- può essere usato da solo in un nodo per 3 motivi:
  - rappresenta l'intersezione di una `highway` con un marciapiede `kerb` (lunghezza nulla)
  - per indicare maggiori informazioni sul tipo di `kerb` (`kerb=lowered` ...) quando sono in un elemento che indica per esempio una fermata del bus o di un attraversamento pedonale

il nodo contenente il tag kerb fa parte di una way di tipo

Ho trovato un esempio di kerb high che è una way che rappresenta questo:



Quindi bisogna stare attenti ad usarlo

I dati sono abbastanza densi:

```
[out:json];
```

Questo è il primo caso in cui kerb viene usato come valore del tag `barrier`.

```

relation(44915) → .milano;
.milano map_to_area → .areaMilano;

(
nwr(area.areaMilano)["barrier"~"kerb"];
nwr(around.milano:1500)["barrier"~"kerb"];
) → .risultato;

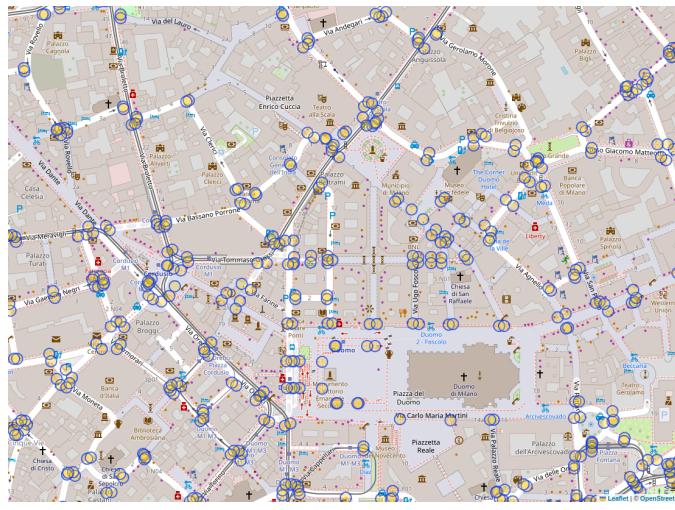
.risultato out geom;

```

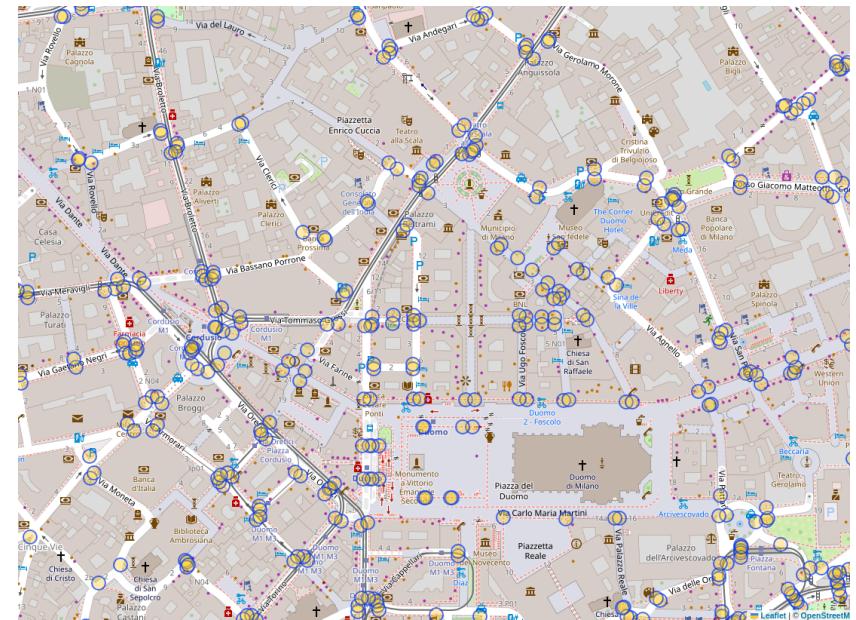
Di questi kerb risultanti si possono prendere solo quelli adatti alle sedie a rotelle utilizzando il secondo modo:

`kerb~"flush|lowered|no"`

[Documentazione](#)



kerb in generale



Kerb flush o lowered

Tra le due immagini c'è qualche nodo mancante, quei nodi sono quelli in cui il kerb non è adatto per le sedie a rotelle:

```

[out:json];

relation(44915) → .milano;
.milano map_to_area → .areaMilano;

( // kerb adatti
nwr(area.areaMilano)["barrier"~"kerb"]["kerb"~"flush|lowered|no"];
nwr(around.milano:1500)["barrier"~"kerb"]["kerb"~"flush|lowered|no"];
) → .adatti;

( // tutti i kerb
nwr(area.areaMilano)["barrier"~"kerb"];
nwr(around.milano:1500)["barrier"~"kerb"];
) → .tutti;

// differenza per trovare quelli non esplicitamente adatti:
(.tutti; - .adatti;) → .nonAdatti;

.nonAdatti out geom;

```



Kerb non esplicitamente adatti alle sedie a rotelle

Una volta estratti tutti questi dati li ho inseriti in un unico "database" (nient'altro che un file `.json`) per poterli analizzare e classificare per bene.

## 2 Esecuzione Query e Classificazione

Ho quindi iniziato creando uno script in python: `API_executor.py` il cui compito è quello di:

1. leggere i file `.txt` contenenti le query scritte in 1.4
2. chiamare l'API di overpass per eseguire ognuna di queste query (esente da un'API key)
3. immagazzinare i risultati di ciascuna query in un file nominato `nome_query_QL.json`

### 2.1 Primo Approccio per la Classificazione

Questi file sono poi diventati la base per la parte di classificazione. Parte che è stata integrata in uno script `routingProgram.py` che si occupa anche del path finding (che vedremo nel capitolo 3).

In buona sostanza per ogni file `.json` trovato e per ogni oggetto trovato nell'array denominato come `elements` viene creato un oggetto Python di tipo `ElementOSM` estraendo:

- latitudine e longitudine
- vengono aggiunte le coordinate del centroide se l'elemento non è un point ma una way o una relation
- si identifica se questo elemento può generalmente essere considerato come una barriera o un facilitatore basandomi sui suoi tags

```
# Capisco se l'elemento_json è una barriera o un facilitatore in base ai tags
if _è_facilitatore_da_tags(tags):
    return Facilitatore(id_elemento, tipo_elemento, geometria, tags)
else:
    return Barriera(id_elemento, tipo_elemento, geometria, tags)
```

- L'elemento diventa quindi o un facilitatore o una barriera e viene aggiunto ad una lista di elementi che poi la parte di path finding userà per determinare il percorso migliore

### 2.2 Secondo Approccio molto più Flessibile

A seguito della richiesta di dare agli utenti la possibilità di personalizzare il loro profilo esprimendo preferenze su barriere (che riescono facilmente a superare) o su facilitatori (che non sono di loro particolare interesse), ho deciso di cambiare totalmente approccio per quando riguarda la classificazione degli elementi estratti precedentemente.

Ho deciso innanzitutto di creare una nuova tipologia di elemento JSON custom che prescinda dall'essere stato estratto da una query Overpass-QL. Quindi l'ho strutturato in questo modo:

```
{  
    "id": "mock-element-id",  
    "barrieraPer": [  
        "Motoria"  
    ],  
    "facilitatorePer": [],  
    "infrastrutturaPer": [],  
    "autore": "Pippo", ← username dell'utente  
    "ranking": 50,  
    "nome": "barriera-generica",  
    "descrizione": "Questo è un dato di mock dell'utente Pippo",  
    "immagine": null,  
    "elementoOSM": null, ← per un elemento estratto da OSM  
    "coordinateCentroide": {  
        "longitudine": 9.184913,  
        "latitudine": 45.466296  
    }  
}
```

Questa nuova struttura aggiunge innumerevoli benefici:

- per il `routingProgram.py` i dati sono tutti standardizzati e non hanno sorprese
- Esiste una nuova tipologia di elemento, le infrastrutture (panchine, fontanelle, bagni pubblici...)
- ogni elemento ha un nome ed una descrizione per:
  - visualizzare gli elementi più facilmente nella mappa
  - fare in modo che gli utenti possano esprimere preferenze in funzione del nome dell'elemento
- l'autore implica che l'applicazione non si limita solo ad includere i dati estratti precedentemente dalla query OSM ma supporta anche l'aggiunta da parte degli utenti di nuovi elementi
- ranking sarà analizzato dopo nel capitolo 3
- Ho creato un nuovo `Enum` per specificare meglio i possibili problemi di mobilità:

```
class ProblemiMobilità(Enum):  
  
    # Disabilità Fisica  
    MOTORIA = "Motoria"  
  
    # Disabilità Sensoriali  
    VISIVA = "Visiva"  
    UDITIVA = "Uditiva"  
  
    # Altre disabilità  
    MULTIPLE = "Multiple"  
    ALTRE = "Altre"
```

Il programma si comporterà poi in modo diverso in funzione del tipo di disabilità di un utente

- le voci `barrieraPer`, `facilitatorePer` ed `InfrastrutturaPer` servono a dare una classificazione di gran lunga più potente rispetto al primo approccio preso: se prima un elemento poteva essere o solo barriera o solo facilitatore adesso lo stesso elemento può essere un'infrastruttura per un utente con problematiche visive e una barriera per un utente con problematiche motorie ecc...

- qualunque sia il tipo di elemento OSM verranno sempre estratte/calcolate le coordinate del centroide

Ma non è finita qui! Al posto di tenere questa logica all'interno dell'ex `routingProgram.py` del primo approccio ho deciso di spostarla direttamente nello script che si occupa dell'esecuzione delle query Overpass-QL cioè: `data_extractor_from_OSM_script.py`. Il funzionamento è molto simile al `API_executor.py` del capitoletto di prima ma prima di inserire gli elementi in un file direttamente dal risultato della query li uniforma nel formato sopra specificato.

Questo è un esempio di classificazione in funzione dei tag:

```
# kerb (marciapiede/bordo strada)
elif elementoOSM["tags"].get("barrier") == "kerb":
    nome = "Bordo Marciapiede"
    if elementoOSM["tags"].get("kerb") == "raised":
        comeBarrieraPer.append(ProblemiMobilita.MOTORIA)
        descrizione = "Marciapiede con bordo alto, difficile da superare."
    else:
        comeFacilitatorePer.append(ProblemiMobilita.MOTORIA)
        descrizione = "Marciapiede con bordo basso/smussato, facile da superare."
if elementoOSM["tags"].get("tactile_paving") == "yes":
    comeFacilitatorePer.append(ProblemiMobilita.VISIVA)
    descrizione += " Presenza di pavimentazione tattile per non vedenti."
```

Oltre a classificare si inserisce anche il nome ed una descrizione. Come questo `elif` ce ne sono molti altri per classificare gli elementi restanti estratti da OSM.

Il capitolo successivo descriverà il path finding con quest'ultimo approccio di classificazione dei dati raw; partì quindi da quei file `.json` nel nuovo formato standardizzato descritto precedentemente.

## 3 Path Finding e Nuove Funzionalità

Bene ora che ho raccolto i dati e li ho classificati in un formato standard posso finalmente metterli a buon uso definendo:

- coordinate di inizio e di fine del percorso che l'utente finale vuole calcolare
- l'utente stesso che comprende il nome, la disabilità e le preferenze che questo ha
- grandezze in metri di vari buffer (vedi in seguito)

Questi diventano quindi i principali input dello script `routingProgram.py`.

### 3.1 Come Fare l'Effettivo Routing

Creare un programma in grado di poter fare path-finding efficientemente e robustamente sarebbe stato decisamente infattibile specialmente se questo avrebbe dovuto lavorare in locale sulla mappa di OSM.

Così sono andato alla ricerca di numerose API per capire se sarebbero potute essere in grado di fare path-finding sulla mappa di OSM con l'aggiunta di geometrie di ostacoli (le barriere) e way-points (ovvero i facilitatori e le infrastrutture).

La ricerca s'è conclusa una volta trovata [openrouteservice](#) (ORS in breve). Questa API ha molti benefici per questo caso d'uso:

- è totalmente open source il che vuol dire che si può eseguire un istanza, anche modificata, del server in locale! il che rende praticamente illimitate e customizzabili le chiamate all'API.
- aveva dei limiti molto alti di chiamate gratuite il che è perfetto dato che il programma è solo nella fase di development & testing
- il principale vantaggio è che la chiamata oltre ovviamente alle coordinate di inizio e fine include:
  - il campo del "mezzo utilizzato" (in questo caso tutte le chiamate sono per percorsi pedonali)
  - delle geometrie potenzialmente molto complesse da evitare durante la creazione del percorso. Questa è una feature che ben poche altre API avevano. Ed è stata importantissima dato che mi ha consentito di "raggirare" il basso livello di personalizzazione che queste API normalmente hanno.
  - ulteriori way-points intermedi da cui il percorso deve passare (perfetti per i facilitatori e le infrastrutture che l'utente vuole includere)

- Una bellissima documentazione interattiva
- la restituzione del risultato in formato JSON con tanto di indicazioni verbose e `polyline` da decodificare per mostrare il percorso su una qualsiasi mappa (`Folium` nel mio caso)

Nel codice ho quindi implementato la funzione:

```
# chiamata all'API di OpenRouteService per calcolare i percorsi
def chiamataAPIdiORS(inizio, fine, elementi_da_evitare, waypoints, preferenze):
    """
    Calcola uno o più percorsi pedonali usando OpenRouteService:
    - inizio: coordinate d'inizio
    - fine: coordinate dell'arrivo
    - elementi_da_evitare convertiti poi in multipoligoni per la chiamata
    - waypoints: lista di elementi da includere nel percorso
    - preferenze: è proprio un campo per la chiamata (es. "fastest")

    returns una lista di "routes" di cui prendo sempre solo il primo
    """

```

Questi routes sono i segmenti del percorso calcolato e sono più di uno solo in casi di percorsi molto lunghi o in caso di utilizzo di mezzi differenti lungo il percorso. Per le casistiche coperte da questo programma la lista ritornata come risultato dalla chiamata non dovrebbe contenere mai più di un elemento.

## 3.2 Utilizzo Strategico dell'API di ORS

Ora che da 3.1 sono in grado di ottenere un percorso, devo lavorare sulla sua presentazione e sul come ottenere gli input necessari dall'utente per avviare la fase di personalizzazione.

### 3.2.1 Logica Iterativa

Nella prima versione del codice ho adottato una logica iterativa:

1. l'utente viene presentato col percorso standard più veloce possibile che non tiene bada del numero di barriere incontrate o di qualsiasi altro fattore di preferenza dell'utente
2. l'utente, ispezionando il percorso (su cui sono segnalate le barriere e i facilitatori), sceglie quali barriere vuole evitare e quali facilitatori vuole includere lungo il percorso
3. a seguito delle scelte dell'utente viene fatta una nuova chiamata a ORS contenente le scelte sottoforma di waypoints e geometrie da evitare
4. il tutto si ripete fino a che l'utente è soddisfatto e non fa alcuna scelta di inclusione di facilitatori o esclusione di barriere

Illustrazione di un tipico passaggio tra iterazioni di percorsi poco a poco più personalizzati:

```

===== ITERAZIONE 1 =====
Cercando barriere e facilitatori per utente con disabilità: NON VEDENTE...
Trovate 6 barriere e 19 facilitatori sul percorso.
Mappa del percorso salvata in: mappa_percorso.html

===== BARRIERE TROVATE (6) =====
1. Barriera attraversamento (ID: 1722395913) - Tipo: attraversamento, tactile_paving: no
2. Barriera attraversamento (ID: 2609160271) - Tipo: attraversamento, tactile_paving: no
3. Barriera attraversamento (ID: 2923126653) - Tipo: attraversamento, tactile_paving: no
4. Barriera attraversamento (ID: 3455715407) - Tipo: attraversamento
5. Barriera attraversamento (ID: 2951281592) - Tipo: attraversamento, tactile_paving: no, wheelchair: no
6. Barriera attraversamento (ID: 2951281593) - Tipo: attraversamento, tactile_paving: no

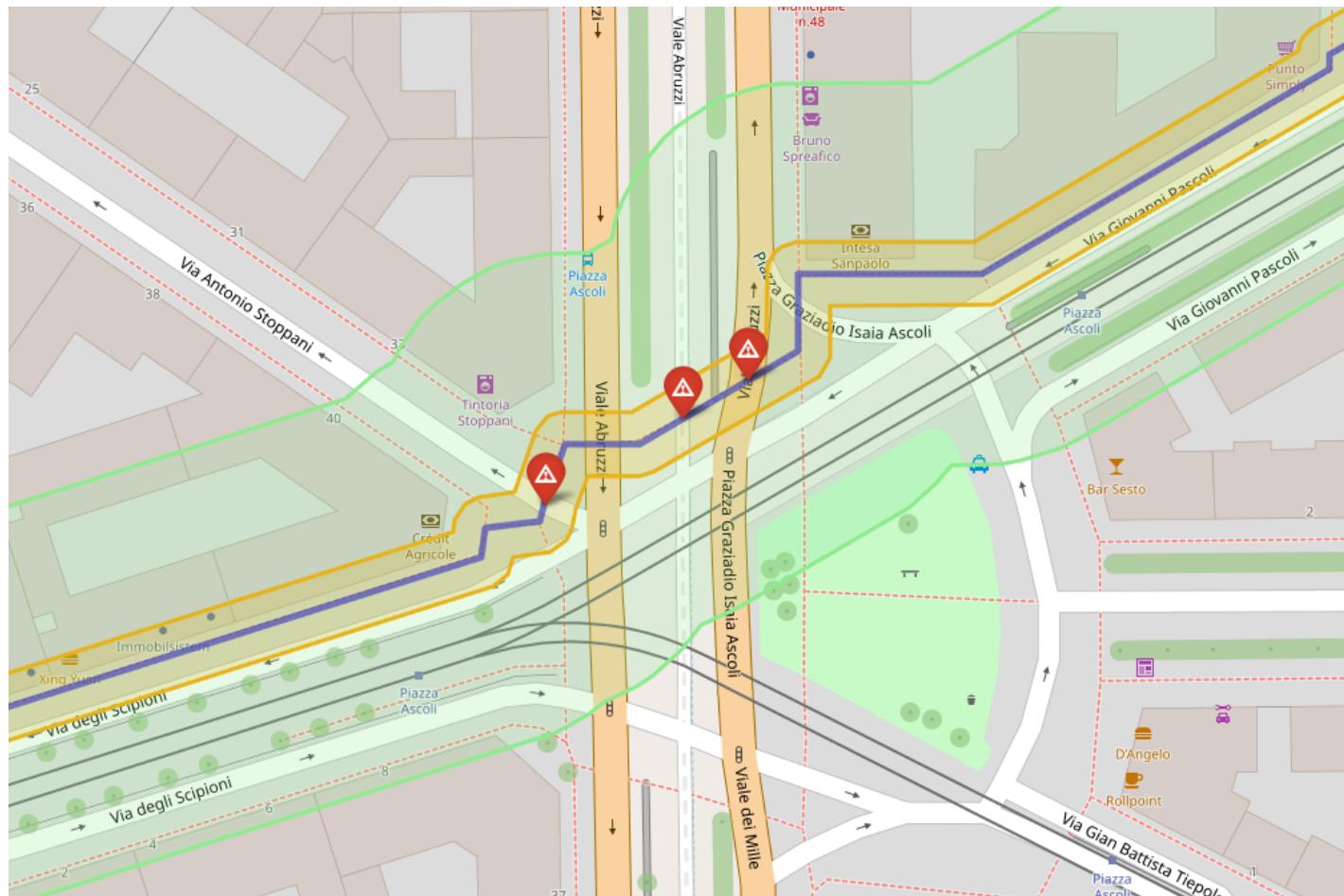
===== FACILITATORI TROVATI (19) =====
1. Facilitatore attraversamento_sonoro (ID: 252640097) - Tipo: attraversamento_sonoro
2. Facilitatore attraversamento_sonoro (ID: 1596494673) - Tipo: attraversamento_sonoro
3. Facilitatore attraversamento_sonoro (ID: 1722395921) - Tipo: attraversamento_sonoro, tactile_paving: no, wheelchair: yes
4. Facilitatore attraversamento_sonoro (ID: 1957305972) - Tipo: attraversamento_sonoro
5. Facilitatore attraversamento_sonoro (ID: 3372962174) - Tipo: attraversamento_sonoro, tactile_paving: no, wheelchair: yes
6. Facilitatore attraversamento_sonoro (ID: 3720223375) - Tipo: attraversamento_sonoro, tactile_paving: no, wheelchair: yes
7. Facilitatore attraversamento_sonoro (ID: 3720223376) - Tipo: attraversamento_sonoro, tactile_paving: no
8. Facilitatore attraversamento_sonoro (ID: 3720223377) - Tipo: attraversamento_sonoro, tactile_paving: no
9. Facilitatore attraversamento_sonoro (ID: 3720223378) - Tipo: attraversamento_sonoro, tactile_paving: no
10. Facilitatore attraversamento_sonoro (ID: 4059821576) - Tipo: attraversamento_sonoro, tactile_paving: no
11. Facilitatore attraversamento_sonoro (ID: 4059821577) - Tipo: attraversamento_sonoro, tactile_paving: no, wheelchair: yes
12. Facilitatore attraversamento_sonoro (ID: 4059821580) - Tipo: attraversamento_sonoro, tactile_paving: no
13. Facilitatore attraversamento_sonoro (ID: 4059821583) - Tipo: attraversamento_sonoro, tactile_paving: no
14. Facilitatore attraversamento_sonoro (ID: 4059825811) - Tipo: attraversamento_sonoro, tactile_paving: no
15. Facilitatore attraversamento_sonoro (ID: 4059825815) - Tipo: attraversamento_sonoro, tactile_paving: no
16. Facilitatore attraversamento_sonoro (ID: 4305611664) - Tipo: attraversamento_sonoro, tactile_paving: no
17. Facilitatore attraversamento_sonoro (ID: 4889789472) - Tipo: attraversamento_sonoro
18. Facilitatore attraversamento_sonoro (ID: 6457361429) - Tipo: attraversamento_sonoro, tactile_paving: no
19. Facilitatore attraversamento_sonoro (ID: 10543168139) - Tipo: attraversamento_sonoro

Seleziona i numeri delle barriere che vuoi evitare (separati da virgola) (Enter per nessuna):
> 4, 2

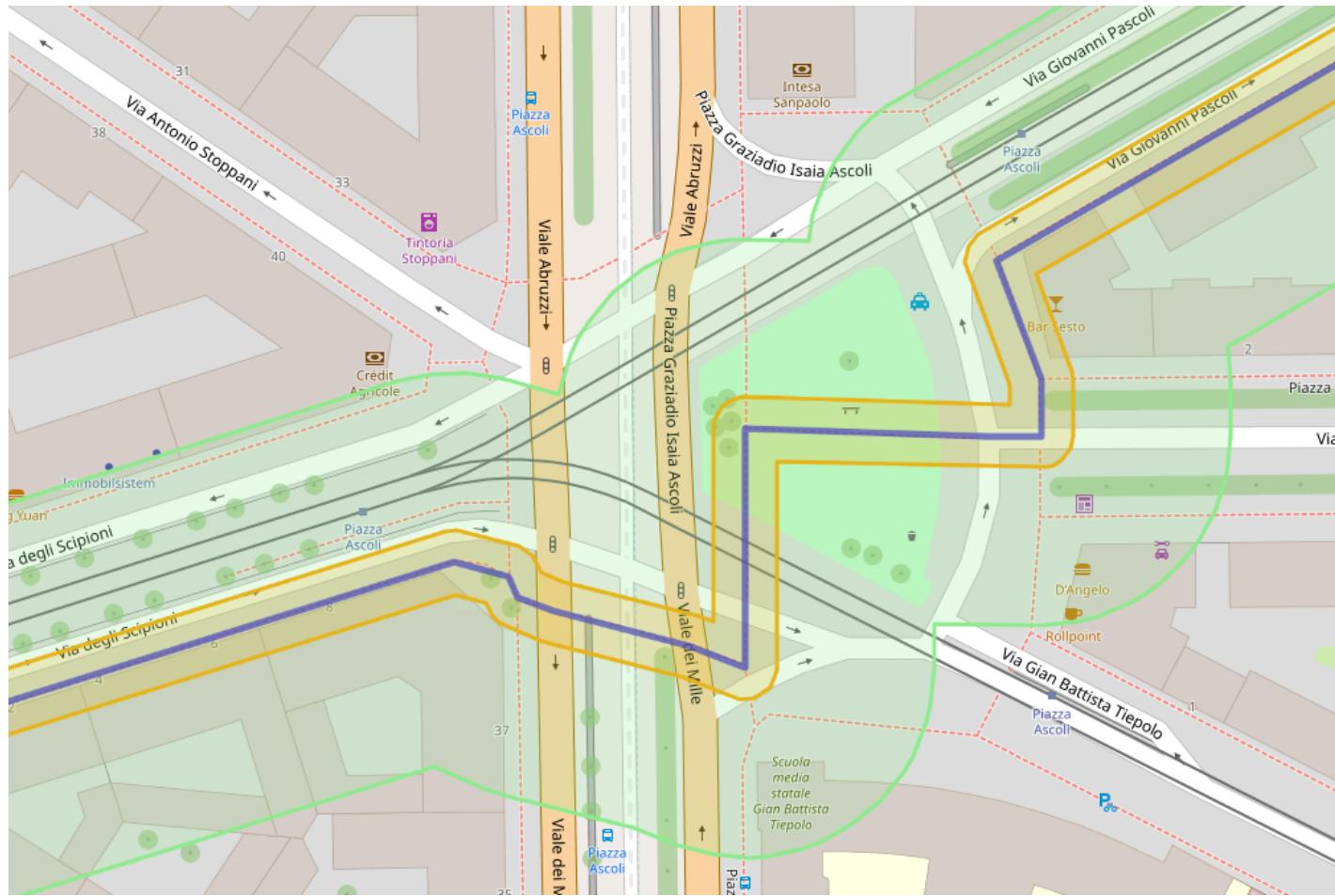
Seleziona i numeri dei facilitatori che vuoi includere (separati da virgola) (Enter per nessuno):
>

```

Menù di selezione delle barriere e dei facilitatori



l'utente non vedente ha deciso di evitare quei 2 attraversamenti pedonali senza pavimento tattile



e il programma nella seconda iterazione ha calcolato il percorso escludendo le due geometrie degli attraversamenti evitati.

Seppur questa metodologia permette di avere il percorso più personalizzato possibile lascia spazio a moltissime potenziali migliorie, prima di tutte è lo scarso livello di automazione.

Questo è infatti stato il motivo principale per il cambio di approccio che ho menzionato in 2.2.

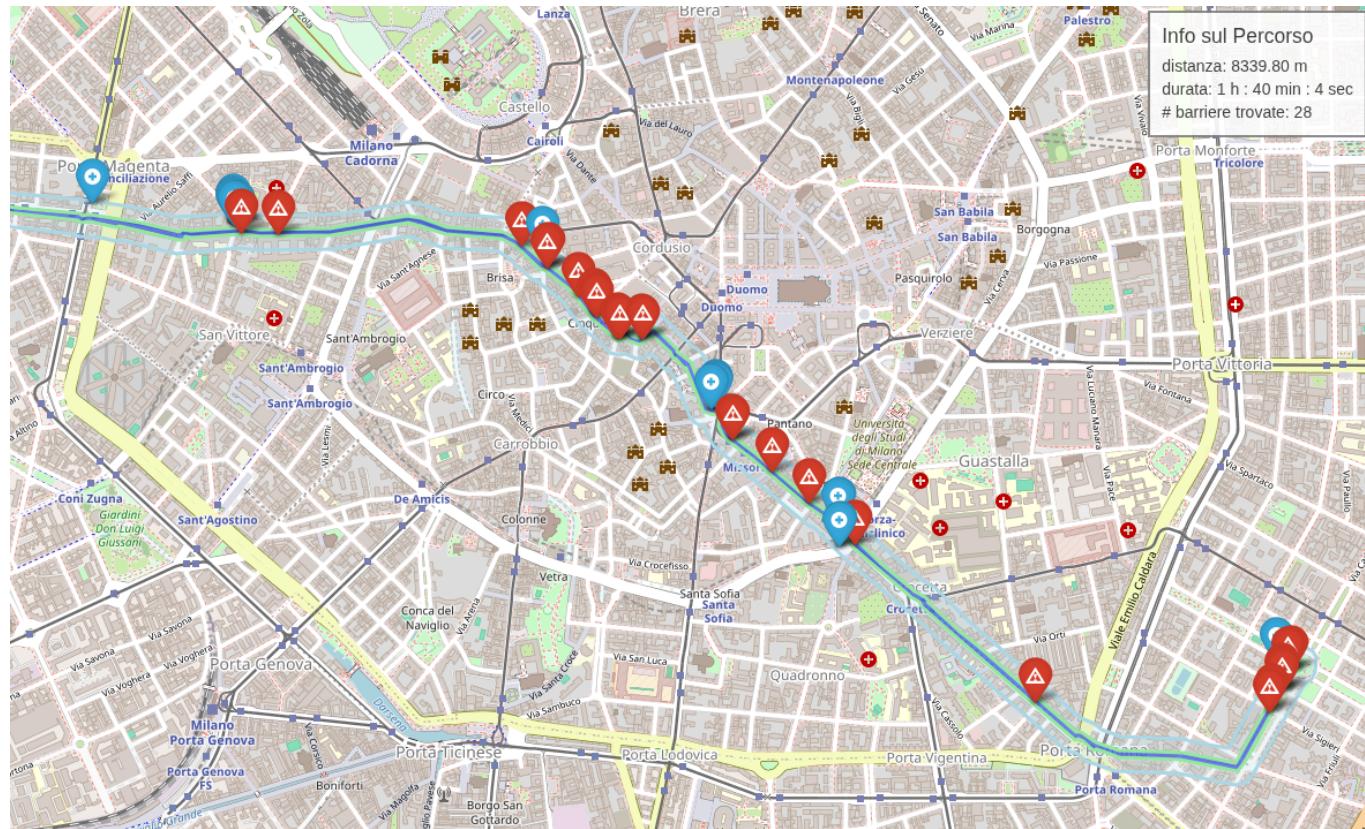
### 3.2.2 Logica Automatica con Preferenze dell'Utente

Questo approccio punta molto sulla personalizzazione automatica del percorso in funzione dell'utente, la sua problematica e le sue preferenze (tutti questi aspetti saranno meglio coperti dopo).

Innanzitutto questo nuovo approccio è costruito sulla base di una logica di classificazione molto più robusta e flessibile di quella precedente (2.1). Questo permette di selezionare le barriere, i facilitatori e le infrastrutture in modo più granulare di prima e più in funzione di quelli che potrebbero essere i reali bisogni dell'utente.

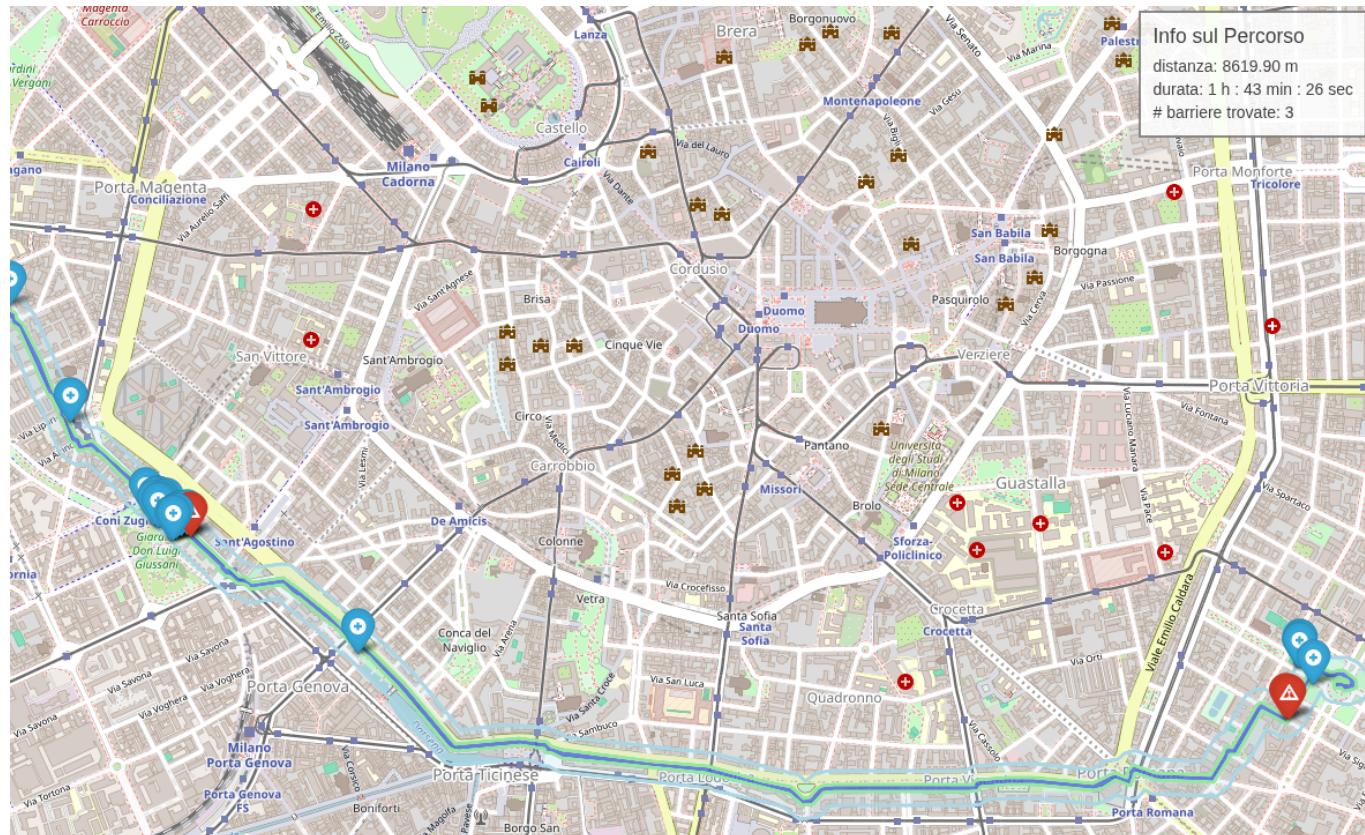
Questa logica automatica trova 2/3 percorsi a seconda della lunghezza senza bisogno di alcun input user-side (tranne ovviamente coordinate di inizio e fine):

1. il primo percorso calcolato è come prima quello più veloce senza considerare alcuna barriera, preferenza o ranking.



Percorso standard di 1h:40min con 28 barriere trovate per l'utente in questione

2. il secondo percorso è invece trovato nella stessa identica metodologia applicata nel primo approccio ma con la selezione automatica di tutte le barriere da evitare. Le iterazioni fatte sono tipicamente 4/5.



Percorso a iterazioni automatiche di 1h:43min (3 minuti in più) con solo 3 barriere trovate! (con 3 iterazioni)

1. i vantaggi di questo nuovo approccio sono sicuramente la velocità e l'automazione. Infatti un utente della prima versione avrebbe dovuto manualmente rimuovere ognuna di quelle 28 barriere per poter arrivare ad un percorso simile a quello calcolato automaticamente in questo caso.
  2. gli svantaggi sono però un numero di chiamate all'API di ORS potenzialmente elevato, anche se questo problema può essere decisamente mitigato dall'utilizzo di un'istanza locale del server
  3. il terzo percorso è invece una chiamata sola che esclude tutte le barriere che la classificazione ha analizzato essere tali da tutta la bounding box contenente il percorso calcolato al punto 1.
    - a. i vantaggi sono dati da un risultato finale che sicuramente è esente da qualsiasi barriera potenzialmente interessante per l'utente che sta calcolando il percorso
    - b. lo svantaggio principale è la chiamata onerosa all'API di ORS che, se il percorso risulta essere troppo lungo, potrebbe lanciare un messaggio di errore. Ancora questa cosa potrebbe essere bypassabile attraverso un'istanza locale modificata del server ORS.
- Infatti per questo percorso così lungo succede proprio questo

```
Errore HTTP nel calcolo del percorso: 413 Client Error:  
Request Entity Too Large for url:  
https://api.openrouteservice.org/v2/directions/foot-walking/json  
Richiesta troppo grande!
```

### 3.3 Importazione dei Dati dal JSON al Programma

Fra le due versioni il codice ha subito grossi cambiamenti. Nella prima versione tutti gli elementi del "database" JSON erano inseriti in un oggetto python. Il tutto è estremamente inefficiente dato che a tutti gli effetti si stava caricando in memoria l'intero DB (l'ho fatto solo per motivi di testing e rapid development).

La seconda versione invece implementa una sorta di DBMS in grado di estrarre dai file di database solo quelli necessari al percorso correntemente calcolato:

1. si calcola il primo percorso e la relativa bounding box ( `bbox` )
2. si iterano tutti gli elementi standardizzati contenuti nei file `.json` e si fanno vari controlli per capire quali elementi caricare in memoria (ovvero il risultato della query di questo DBMS diy):

```
def caricaElementiDaJSON(directory_risultati, bbox, utente):
    ...
    Carica tutti gli elementi dai file JSON che trova nella directory
    "data" che:
        - abbiano il centroide all'interno della bbox specificata
```

- che siano di interesse per l'utente specificato

'''

Per ogni file `.json`:

```
data = json.loads(contenuto)
# Processa i dati JSON
aggiunti = 0
for elemento in data:
    # controllo se l'elemento può essere utile per l'utente
    if utente.interessa(elemento):
        # controllo se rientra nella bounding box
        if bbox[0] <= elemento["coordinateCentroide"]["longitudine"] <= bbox[2]
        and bbox[1] <= elemento["coordinateCentroide"]["latitudine"] <= bbox[3]:
            elemento_osm = Elemento(elemento) # Crea l'elemento OSM
            if elemento_osm:
                elementi.append(elemento_osm) # E lo aggiunge agli altri
                aggiunti += 1
```

Quindi già in questo modo riesco a ridurre notevolmente il numero di elementi OSM (e non) con cui lavorare. Ma non è finita qui, dato che ancora la stragrande maggioranza di elementi della `bbox` non sono d'interesse all'utente poiché questi non sono in prossimità del percorso che andrà effettivamente a percorrere.

Bisogna quindi fare una seconda scrematura stavolta prendendo in input anche il percorso vero e proprio che l'utente sta considerando. È da qui che deriva la necessità di calcolare sempre un percorso iniziale senza badare ad alcuna barriera, facilitatore o infrastruttura

Nasce quindi il bisogno di creare 3 buffer di dimensioni personalizzabili generati come geometrie attorno al percorso calcolato (ciascuno dei buffer per barriere, facilitatori e infrastrutture con grandezze crescenti in quest'ordine).

Una volta creati questi buffer posso finalmente iterare gli elementi estratti precedentemente per vedere se cadono o meno dentro il rispettivo buffer:

```
def trovaElementiSulPercorso(
    self,
    elementi_caricati_dal_db_che_interessano_a_utente,
    utente
):
    self.barriere_trovate = []
    self.facilitatori_trovati = []
    self.infrastrutture_trovate = []

    for elemento in elementi_caricati_dal_db_che_interessano_a_utente:
        # Verifica cosa l'elemento è per l'utente e se è nel rispettivo buffer

        if elemento.per(utente) == TipoElemento.FACILITATORE and
            self.isNelBuffer(elemento, "facilitatori"):
            self.facilitatori_trovati.append(elemento)

        elif elemento.per(utente) == TipoElemento.BARRIERA and
            self.isNelBuffer(elemento, "barriere"):
            self.barriere_trovate.append(elemento)

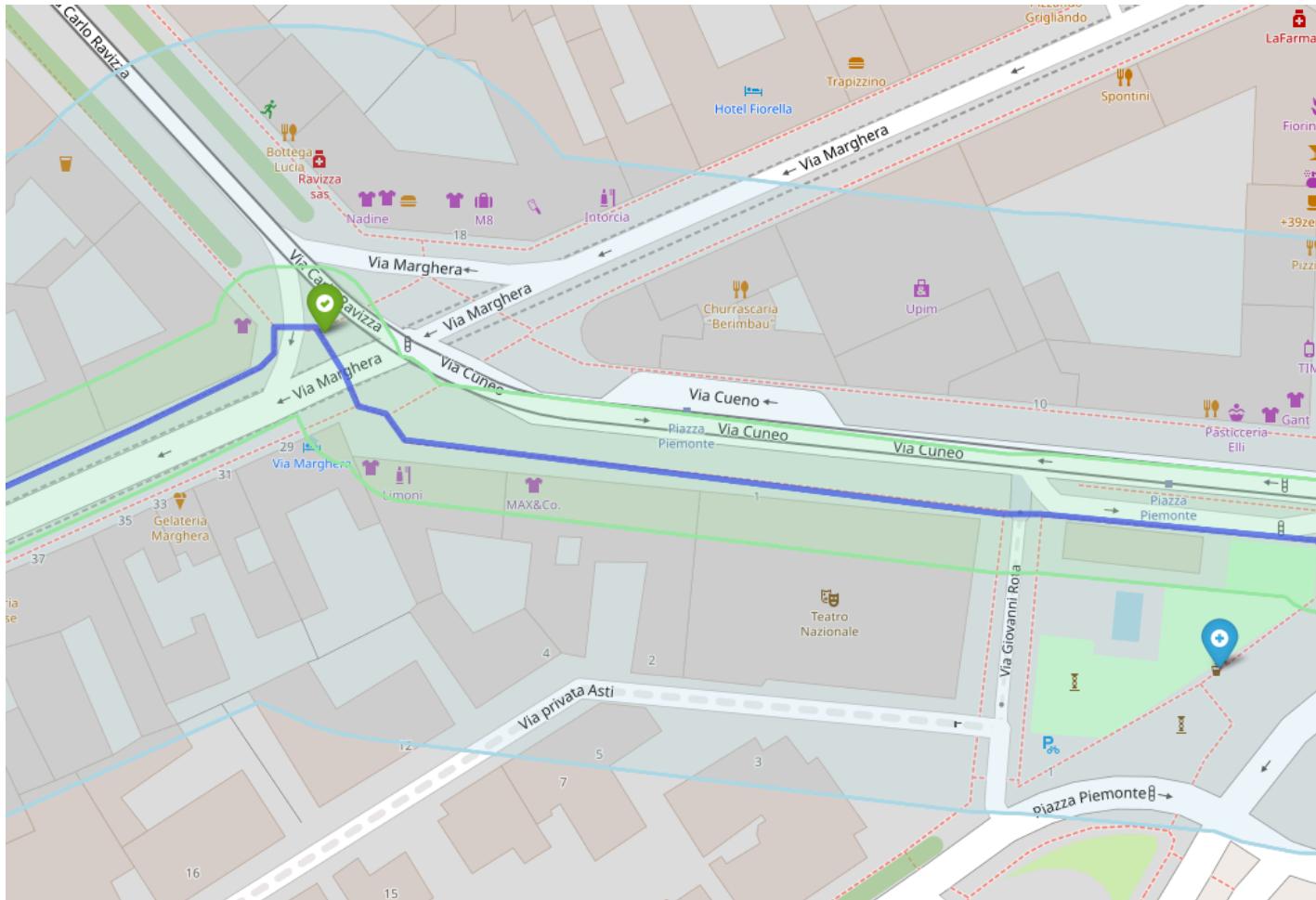
        elif elemento.per(utente) == TipoElemento.INFRASTRUTTURA and
            self.isNelBuffer(elemento, "infrastrutture"):
            self.infrastrutture_trovate.append(elemento)

    else:
```

continue

```
return self.barriere_trovate, self.facilitatori_trovati,  
self.infrastrutture_trovate
```

Ora finalmente ho i dati che mi servono per mostrare la mappa all'utente contenente appunto i centroidi di tutti gli elementi che sono stati scremati da quelli precedentemente caricati dal DB.



Questa è un'illustrazione di come i buffer vengono creati ed utilizzati a partire dal percorso blu calcolato. Il buffer verde è per la ricerca dei facilitatori mentre quello azzurro ricerca le infrastrutture. Il buffer per le barriere esiste ma non è neanche stato disegnato per motivi grafici.

### 3.4 Autore-Ranking & Preferenze Utente

Fino ad ora no ho mai parlato di come il programma decide come un certo elemento OSM (o non) ( $E$ ) possa essere di interesse o meno per un determinato utente ( $U$ ).

Questa funzionalità è implementata mediante 2 metodi delle rispettive classi `Utente` e `Elemento`:

- Nella classe `Elemento` il metodo `E.per(U)` ritorna l'enum `TipoElemento`

```
def per(self, utente):  
    """  
        Ritorna cosa self è per l'utente  
        (barriera / facilitatore / infrastruttura)  
    """  
  
    if utente.problema.value in self.barriera_per:  
        return TipoElemento.BARRIERA  
    elif utente.problema.value in self.facilitatore_per:  
        return TipoElemento.FACILITATORE  
    elif utente.problema.value in self.infrastruttura_per:  
        return TipoElemento.INFRASTRUTTURA  
    else:  
        return None
```

Questa classificazione era già stata fatta nello script `data_extractor_from_OSM_script.py` (vedi 2.2)

- Invece nella classe `Utente` è presente il metodo `U.interessa(E)`:

```
def interessa(self, elemento):  
  
    name_elemento = elemento.get("name")
```

```

# gestione delle preferenze di un utente
if name_elemento in self.tipologia_di_elemento_da_includere_sempre:
    return True
if name_elemento in self.tipologia_di_elemento_da_evitare_sempre:
    return False

# estrapolo classificazione
barriera_per = elemento.get("barrieraPer", [])
facilitatore_per = elemento.get("facilitatorePer", [])
infrastruttura_per = elemento.get("infrastrutturaPer", [])

# vedo se un elemento è pertinente e poi ritorno True solo in base al
# ranking (nient'altro che un valore di priorità)
if str(self.problema) in barriera_per or str(self.problema) in
    facilitatore_per or str(self.problema) in infrastruttura_per:
    # se l'elemento è inerente ed è stato creato dall'utente
    if elemento.get("autore") == self.nickname:
        # allora lo includo poiché sicuramente per lui è interessante
        # (dato che è stato lui a crearlo)
        return True

    # se l'elemento ha ranking di 100 allora è sicuramente interessante
    if elemento.get("ranking") == 100:
        return True

    # altrimenti ritorno True in funzione della probabilità
    return elemento.get("ranking") >= random.randint(0, 100)

return False

```

Qui le cose si fanno decisamente più interessanti dato che ora si considera sia il nome dell'utente che il ranking dell'elemento che una componente randomica!

### 3.4.1 Ranking e Nome Utente

Dato che si è voluto dare la possibilità agli utenti di interagire con la mappa per mappare degli elementi che loro reputano utili o interessanti, è contemporaneamente sorto il bisogno di dare un ranking a questi nuovi elementi per evitare che mal intenzionati o errori accidentali causino l'esistenza di apparenti barriere/facilitatori che in realtà non dovrebbero esistere.

Dettagli e funzionamento del ranking:

- Ogni elemento inizia con un ranking percentuale compreso tra 0% e 25%. Questo valore indica la probabilità che un utente con una disabilità corrispondente visualizzi l'elemento se il suo percorso lo include.
- Il numero del ranking in sé rappresenta la probabilità (infatti va da 0 a 100) con cui un utente:
  - affine all'elemento (cioè di suo potenziale interesse),
  - differente da quello che ha creato l'elemento,
  - e che ha cercato un percorso il cui buffer fa rientrare l'elemento rispettivo
- Un elemento creato dall'utente X ha priorità massima di visualizzazione per lo stesso X. Questo significa che, quando X percorrerà un itinerario che include tale elemento, quest'ultimo gli comparirà sempre come una barriera indipendentemente dal ranking che il sistema gli ha assegnato.
- Il ranking di un elemento può essere modificato in base al feedback degli utenti:
  - Al termine di un percorso, un utente diverso dal creatore può confermare o smentire l'utilità di uno o più elementi tramite una notifica di validazione dell'applicazione.
  - Utenti dedicati alla verifica degli elementi user-created possono influenzare direttamente il ranking nel database.

## Differenza tra OSM e User-Created:

I dati estratti da OSM sono inizialmente considerati affidabili, con un ranking predefinito del 100% (e quindi visibili a tutti gli utenti). Questo ranking può essere abbassato rapidamente se gli utenti, tramite il feedback post-percorso, segnalano inesattezze.

Quando un utente crea un nuovo elemento, il campo del database "elementoOSM" viene impostato su `null` (vedi struttura in 2.2).

Gli altri campi, in particolare le coordinate geografiche, vengono calcolati automaticamente se l'elemento è importato da OSM. Se l'elemento è creato dall'utente, le coordinate vengono specificate tramite un'interfaccia grafica.

Esempio:

L'utente Pippo crea la barriera per le problematiche motorie di nome barriera-generica.

A questa barriera è inizialmente stato dato un ranking di 10, ma col tempo si è alzato a 50.

Adesso il 50% degli utenti che non sono Pippo e che non hanno come preferenze di barriere il nome "barriera-generica" vedono il 50% delle volte l'elemento creato da Pippo se i loro percorsi passano di lì.

Invece l'utente Pippo e tutti gli utenti che hanno come preferenza di barriera il nome "barriera-generica" vedono sempre l'elemento indipendentemente dal ranking che questo ha.

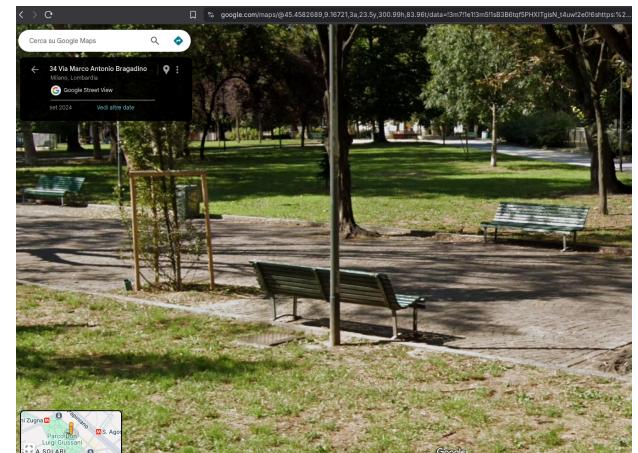
## 3.4.2 Funzionalità Aggiuntive

Per facilitare l'utente circa la visione degli elementi mostrati sulla mappa ho deciso di generare nel pop-up dell'elemento un link google street view impostato sulle coordinate dell'elemento.

Se in quel punto esatto non è presente nessuna immagine google cercherà di prendere l'immagine più vicina fino ad un raggio di 20 metri.



Pop-up con link street view



google street view

Nei pop-up sono anche presenti nomi, descrizioni e identificativo dell'elemento; non il suo autore o ranking.

## 4 Conclusioni

### 4.1 Risultati Ottenuti

#### Estrazione e Classificazione Dati OSM

- Implementazione di un sistema di query Overpass-QL che ha permesso l'estrazione di oltre 20.000 elementi dall'area metropolitana di Milano
- Sviluppo di un algoritmo di classificazione automatica che categorizza gli elementi in barriere, facilitatori e infrastrutture in base al tipo di disabilità
- Creazione di un formato standardizzato JSON che uniforma tutti gli elementi estratti, migliorando l'efficienza di elaborazione

#### Sistema di Routing Personalizzato

- Integrazione con l'API OpenRouteService per il calcolo di percorsi pedonali ottimizzati
- Implementazione di un sistema di buffer dinamici che riduce il carico computazionale filtrando solo gli elementi rilevanti per ciascun percorso

- Sviluppo di un algoritmo iterativo automatico che genera percorsi progressivamente più accessibili senza richiedere intervento manuale dell'utente

## Funzionalità Innovative

- Sistema di ranking probabilistico che gestisce la qualità e l'affidabilità degli elementi user-generated
- Possibilità per gli utenti di contribuire alla mappatura di nuove barriere e facilitatori
- Integrazione con Google Street View per la visualizzazione contestuale degli elementi

## 4.2 Impatto e Applicabilità

Il sistema sviluppato affronta concretamente le sfide dell'accessibilità urbana in vista delle Olimpiadi Milano-Cortina 2026. La capacità di personalizzazione automatica del routing in base alle specifiche esigenze dell'utente rappresenta un significativo passo avanti rispetto alle soluzioni esistenti, che tipicamente offrono un approccio "one-size-fits-all".

### Vantaggi Principali:

- **Scalabilità:** Il sistema può essere esteso ad altre città semplicemente aggiornando le query OSM
- **Personalizzazione:** Ogni utente riceve percorsi ottimizzati per le proprie specifiche esigenze
- **Community-driven:** Gli utenti possono contribuire attivamente al miglioramento della mappatura
- **Efficienza:** L'approccio automatico velocizza drasticamente l'interazione utente-computer

## 4.3 Limitazioni e Sfide

### Dipendenza dalla Qualità dei Dati OSM

La completezza e accuratezza dei risultati dipende fortemente dalla qualità della mappatura di OpenStreetMap nell'area di interesse. Alcune zone potrebbero presentare dati incompleti o obsoleti.

### Limitazioni dell'API OpenRouteService

Per percorsi molto lunghi o con un numero elevato di geometrie da evitare, l'API può restituire errori di "Request Entity Too Large", limitando la funzionalità in alcuni casi d'uso.

### Gestione del Ranking User-Generated

Il sistema di ranking probabilistico, pur innovativo, richiede un numero significativo di utenti attivi per essere efficace nel distinguere tra elementi validi e non validi.

## 4.4 Sviluppi Futuri

### Miglioramenti Tecnici

- **Istanza locale di OpenRouteService:** Implementazione di un server personalizzato per superare i limiti dell'API pubblica
- **Machine Learning:** Integrazione di algoritmi di apprendimento automatico per migliorare la classificazione automatica degli elementi
- **Ottimizzazione delle performance:** Implementazione di caching e pre-calcolo per ridurre i tempi di risposta

### Estensioni Funzionali

- **Integrazione con trasporti pubblici:** Inclusione di informazioni sull'accessibilità di mezzi pubblici, fermate e stazioni (come il sito dell'ATM o altre API fornite dal comune di Milano)
- **Feedback in tempo reale:** Sistema di segnalazione immediata di problemi o modifiche temporanee alla viabilità

### Validazione e Deployment

- **Test su larga scala:** Validazione del sistema con gruppi di utenti con diverse disabilità
- **Partnership istituzionali:** Collaborazione con enti locali e organizzazioni per la disabilità per il deployment operativo
- **Integrazione con le infrastrutture olimpiche:** Mappatura specifica delle venue e dei percorsi olimpici

## 4.5 Contributo Reale

L'approccio sviluppato dimostra come sia possibile combinare tecnologie open source, dati della comunità e algoritmi personalizzati per creare soluzioni concrete ai problemi di accessibilità urbana, fornendo uno strumento prezioso per supportare la mobilità inclusiva durante i Giochi Olimpici Milano-Cortina 2026 e oltre.

## 5. Riferimenti e Repo Github

- **Repository GitHub:** <https://github.com/Alberto-Pini-Polimi/Progetto>
- **Documentazione tecnica:** Specifiche implementative e guide d'uso presenti sulla repo
- **Dataset:** Query OSM e risultati strutturati per le esecuzioni delle query su Milano anch'essi presenti nella repo

Provvederò a caricare nella consegna anche una zip della repo in locale contenente anche la mia API KEY per facilitare l'esecuzione del programma alla professoressa Fugini.