




OpenStreetMap-Neo4j

OpenStreetMap graph database

Antonio Pelusi

Matricola - 182267

Panoramica

OpenStreetMap-Neo4j è un programma da linea di comando che permette di creare, rimuovere e localizzare punti di interesse, insieme ai relativi percorsi.

Permette, tra le diverse funzionalità, di calcolare i cammini minimi tra due punti di interesse.

Istruzioni

Installazione dipendenze

- **Neo4j Desktop:** <https://neo4j.com/download/>
- **Python:** <https://www.python.org/downloads/>
- **Py2neo** `pip install py2neo`
- **APOC:** (guida al download del dataset e di questo plugin nella prossima sezione)

Preparazione e avvio

- Avviare Neo4j Desktop
- Scaricare il sample project **openstreetmap** (versione 4.4.14) e scegliere come password "openstreetmap"
- Aggiungere il plugin **APOC**: Neo4j Desktop -> click sul database appena aggiunto -> Plugin -> APOC -> Installa (è necessario il riavvio del database)
- Muoversi nella directory `/openstreetmap-neo4j`
- Eseguire il programma: `python3 openstreetmap-neo4j.py`
- **Troubleshooting:** Potrebbe presentarsi un errore alla connessione al database durante l'avvio del programma. Per risolvere, basterà ripristinare la password del database a "openstreetmap".

Per l'utilizzo di **OpenStreetMap-neo4j**, si rimanda all'ultima sezione della documentazione (**Test guidato**).

Specifiche

Il programma, scritto in Python, utilizza Py2neo per stabilire una connessione con Neo4j.

Il programma viene eseguito su linea di comando. All'avvio (e dopo ogni operazione) verrà mostrato il menù delle funzionalità. L'utente, inserendo il numero relativo al comando scelto, potrà eseguire la funzione desiderata.

Dataset

Il dataset (<https://github.com/neo4j-graph-examples/openstreetmap>) contiene tutti i punti di interesse presenti nei dintorni di Central Park, New York.

Ogni punto di interesse è salvato in un nodo di tipo *PointOfInterest*, i quali sono localizzabili tramite coordinate e hanno un determinato tipo (ristorante, caffè, statua ecc...).

Ogni incrocio è salvato in un nodo di tipo *Intersection*, anch'essi localizzabili tramite coordinate.

I percorsi tra i diversi nodi (*PointOfInterest* o *Intersection*) sono salvati in relazioni chiamati *Route*.

Comandi

L'utente può interagire con il database tramite l'uso delle seguenti funzioni:

Comando	Descrizione
0	Esci da mtgDB
1	Aggiungi un nuovo Point Of Interest
2	Rimuovi un Point Of Interest già esistente
3	Aggiungi un nuovo Route
4	Rimuovi un Route già esistente
5	Cerca i Point Of Interest vicini ad un posto
6	Cerca e localizza un Point Of Interest tramite nome
7	Filtra i Point Of Interest per tipo
8	Cerca il cammino minimo tra due Point Of Interest
9	Elenca tutti i Route disponibili da un Point Of Interest

Processo di sviluppo

Il codice è strutturato in tre sezioni principali, **inizializzazione**, **funzioni** e **start**.

Nella sezione **inizializzazione** verrà stabilita la connessione con Neo4j tramite Py2neo.

Nella sezione **funzioni** sono presenti tutte le funzioni utilizzate da Neo4j per la gestione del dataset.

Nella sezione **start** partirà l'esecuzione del programma, in cui l'utente potrà chiamare la funzione desiderata.

Inizializzazione

In questa sezione viene inizializzata la connessione con Neo4j:

```
osm = Graph("bolt://localhost:7687", auth=("neo4j", "openstreetmap"))
```

Dopo di che, procedo con il creare un indice sul nome dei Point Of Interest, in modo tale da velocizzare l'esecuzione delle query.

L'indice verrà creato solo nella prima esecuzione del programma, per evitare un inutile overhead:

```
try:
    osm.run("CREATE INDEX ON:PointOfInterest(Name)")
except:
    pass
```

Funzioni

Le funzioni sviluppate possono essere, divise in due categorie:

- **Funzioni di interfaccia:** funzioni che verranno eseguite esclusivamente per tenere un certo decoro estetico al programma;
- **Funzioni di interazione con MongoDB:** funzioni che implementano le query per MongoDB, eseguite con l'ausilio dei metodi messi a disposizione da PyMongo.

Funzioni di interfaccia:

print_logo()

Stampa il logo del programma. Viene eseguita solo all'avvio e alla chiusura del programma.

print_menu()

Stampa il menu dei comandi esistenti. Viene eseguita dopo ogni operazione effettuata.

clear_terminal()

Pulisce il contenuto del terminal prima di eseguire un nuovo comando. Permette di avere una interfaccia sempre pulita con il focus sul menù e sull'output della funzione precedentemente chiamata.

Funziona sia su tutti i sistemi operativi desktop grazie all'uso dell'operatore "|" e della libreria os presente in Python, grazie alla quale è possibile passare direttamente alla linea di comando il comando corretto tra `clear` (per Linux e MacOS) e `cls` per Windows.

Di seguito il comando completo:

```
os.system('cls|clear')
```

Funzioni di interazione con MongoDB:

1: add_POI()

Crea un nuovo nodo di tipo *Point Of Interest*, il quale ha un nome, una latitudine, una longitudine e un tipo.

```
osm.run("""
    CREATE (p:PointOfInterest:OSMNode {name: $name, type: $type, location:
        point({srid: 4326, x: $lon, y: $lat}), lon: $lon, lat: $lat})
    RETURN p.name AS name, p.type AS type, p.location.x AS X, p.location.y AS Y
    """,
    parameters={'name': name, 'type': type, 'lon': lon, 'lat': lat}).to_table()
```

2: remove_POI()

Rimuove un nodo di tipo *PointOfInterest* dal database.

```
osm.run("""
    MATCH (p:PointOfInterest)
    WHERE p.name = $name
    DELETE p
    """, parameters={'name': name})
```

3: add_Route()

Mette in relazione due nodi creando un percorso (*Route*), il quale ha come proprietà la sua lunghezza (*distance*).

```
osm.run("""
    MATCH (source:PointOfInterest), (dest:PointOfInterest)
    WHERE source.name = $source
    AND dest.name = $dest
    CREATE (source)-[:ROUTE{distance: $dist}]->(dest)
    CREATE (source)<[:ROUTE{distance: $dist}]->(dest)
    """, parameters={'source': source, 'dest': dest, 'dist': dist})
```

4: remove_Route()

Rimuove il percorso (*Route*) tra due nodi.

```
osm.run("""
    MATCH (source:PointOfInterest)<-[r:ROUTE]->(dest:PointOfInterest)
    WHERE source.name = $source
    AND dest.name = $dest
    DELETE r
    """, parameters={'source': source, 'dest': dest})
```

5: list_POI()

Elenca tutti i *Point Of Interest* che sono presenti in un certo range in linea d'aria, e ne calcola i cammini minimi, ordinandoli dal più vicino al più lontano.

La distanza in linea d'aria è calcolata utilizzando la funzione **point.distance(source, dest)** utilizzando latitudine e longitudine, mentre calcola i cammini minimi utilizzando l'algoritmo di Dijkstra sulle lunghezze dei *Route* attraverso l'apposita funzione appartenente al plugin APOC. In un applicativo reale, questa funzione rappresenterebbe una ricerca di un luogo per nome.

In questa query sono state utilizzate non solo due funzioni non viste a lezione (apoc.algo.dijkstra() e point.distance()), ma anche due nuovi comandi:

- **CALL** per eseguire una funzione;
- **YIELD** per "trattenere" gli output della funzione chiamata.

```
osm.run("""
  MATCH (source:PointOfInterest), (dest:PointOfInterest)
  WHERE source<>dest
  AND source.name = $name
  AND dest.name <> "pitch"
  AND point.distance(source.location, dest.location) <= $dist
  CALL apoc.algo.dijkstra(source, dest, 'ROUTE', 'distance') YIELD weight
  RETURN dest.name AS name, dest.type AS type, round(weight) AS distance
  ORDER BY distance ASC
  LIMIT 10
""", parameters={'name': name, 'dist': dist}).to_table()
```

6: locate_POI()

Localizza un *Point Of Interest* recuperando le sue coordinate. In un applicativo reale, questa funzione sarebbe alla base di ogni query di ricerca.

```
osm.run("""
  MATCH (p:PointOfInterest)
  WHERE p.name = $name
  RETURN p.name AS name, p.type AS type, p.location.x AS X, p.location.y AS Y
""", parameters={'name': name}).to_table()
```

7: filter_POI()

Filtra tutti i *Point Of Interest* di un certo tipo presenti in un determinato range in linea d'aria. In un applicativo reale, questa funzione è paragonabile alla funzione "near me" di Google Maps.

```
osm.run("""
    MATCH (p1:PointOfInterest), (p2:PointOfInterest)
    WHERE p2.type = $type
    AND p1.name = $name
    AND p2.name <> "pitch"
    AND point.distance(p1.location, p2.location) <= $dist
    RETURN p2.name AS name, p2.type AS type, p2.location.x AS X, p2.location.y
           AS Y
    LIMIT 10
""", parameters={'name': name, 'type': type, 'dist': dist}).to_table()
```

8: sp_POI()

Calcola il cammino minimo tra due punti. In un applicativo reale, permette di calcolare il miglior percorso per un navigatore.

La seguente query stampa in un formato tabellare il punto di partenza, il punto di arrivo e la distanza calcolata sommando tutte le lunghezze dei percorsi scelti dall'algoritmo di Dijkstra.

```
osm.run("""
    MATCH (source:PointOfInterest),(dest:PointOfInterest)
    WHERE source.name = $source
    AND dest.name = $dest
    CALL apoc.algo.dijkstra(source, dest, 'ROUTE', 'distance') YIELD weight
    RETURN source.name, dest.name, round(weight) AS distance;
""", parameters={'source': source, 'dest': dest}).to_table()
```

La seguente query invece stampa il percorso da seguire per spostarsi tra i due *Point Of Interest*. È stata aggiunta una modifica tramite espressione regolare del percorso, con la quale verranno escluse dalla stampa le proprietà dei nodi e dei percorsi.

```
osm.run("""
    MATCH (source:PointOfInterest),(dest:PointOfInterest)
    WHERE source.name = $source
    AND dest.name = $dest
    CALL apoc.algo.dijkstra(source, dest, 'ROUTE', 'distance') YIELD path
    RETURN path;
""", parameters={'source': source, 'dest': dest}).to_subgraph()
```

```
path = re.sub(r'({[^}]*})?[{']', '', path)
print(path)
```


9: list_Route()

Elenca tutti i percorsi che passano per un determinato *Point Of Interest*.

In un applicativo reale, questa funzione sarebbe utile per mostrare le frecce direzionali in una mappa interattiva come Google Street View.

```
osm.run("""
    MATCH routes=(source:PointOfInterest{name: $name})-[:ROUTE]-()
    RETURN routes
""", parameters={'name': name}).to_table()
```

Start

Questa sezione fa da main al programma. Infatti implementa uno switch (attraverso un sistema `if-elif-else`) per controllare e chiamare la funzione scelta dall'utente.

Ogni funzione avrà il suo `if`, mentre l'`else` (che fa da caso *default* dello switch) si occuperà di avvisare l'utente che non vi è alcun comando associato al valore erroneamente inserito.

Indifferentemente dal caso dello switch, verrà pulito il terminale per dare un effetto "applicazione interattiva" (se pur con le limitazioni indotte dalla linea di comando), e verrà controllato lo stato del database (non verrà eseguita la funzione se il database non è stato precedentemente caricato).


Test guidato di mtgDB

Segue una lista di comandi inseribili in ordine per testare le funzionalità del programma sviluppato e i suoi punti critici.

Nel seguente esempio verrà aggiunto un nuovo Point Of Interest all'interno del database, insieme ai relativi collegamenti con altri nodi. Verranno poi effettuate diverse operazioni, tra cui la ricerca di cammini minimi.

Inoltre fare attenzione ad inserire correttamente le lettere maiuscole e minuscole, in quanto l'intero programma è case-sensitive.

- Seguire le istruzioni presenti nell'apposita sezione per l'installazione di Neo4j, di Python e del plugin APOC;
- Avviare Neo4j e avviare il database openstreetmap;
- Eseguire openstreetmap-neo4j.py. Verrà mostrato il logo e il menù. Se risulta esserci un problema di connessione a Neo4j, ripristinare la password del database inserendo nuovamente "openstreetmap" da Neo4j Desktop;
- Essendo questa la prima esecuzione del programma, verrà creato automaticamente l'indice sui nomi dei *Point Of Interest*;
- Creiamo un nuovo *Point Of Interest* utilizzando la funzione **1** inserendo come nome "Test", come tipo "statue", come latitudine 40.7931737 e come longitudine -73.9527528. Il nuovo nodo appena creato è una statua chiamata test con posizione interna al Central Park di New York;
- Per cercare e localizzare il nodo appena inserito, utilizziamo la funzione **6** inserendo il nome "Test";
- Procedo con il collegare il nodo appena creato con un altro nodo già esistente e collegato con la rete utilizzando la funzione **3** inserendo poi i due nodi da collegare, prima "Test" e poi il nodo già esistente "The Pond", dandogli una distanza fittizia di 35 metri;
- Provo ora a cercare il cammino minimo tra il nodo "Test" e un altro nodo già esistente, come "Overlook Rock" utilizzando il comando **8** e inserendo il nome dei due nodi. Verrà mostrata sia la distanza che il percorso da seguire. I nodi del tipo _XXXX (con al posto delle X dei numeri) sono gli incroci tra le strade. Come possiamo notare, per arrivare da un nodo all'altro, si passerà per il nodo "The Pond" precedentemente collegato con "Test";
- Eliminiamo ora il *Route* precedentemente creato con la funzione **4** passando al programma i due nomi "Test" e "The Pond";
- Eliminiamo anche il *Point Of Interest* precedentemente creato utilizzando la funzione **2** inserendo il nome "Test". Eliminando il nodo senza eliminare le sue relazioni sarebbe stato comunque possibile grazie al comando DETATCH;

- 
- Utilizzando la funzione **5**, inserendo il nome “**The Pond**” e come distanza 1000 (1 kilometro) verranno elencati i dieci *Point Of Interest* più vicini che rientrano nel range aereo della distanza inserita, in questo caso 1000 metri;
 - Se volessimo cercare tutte le fontane nel raggio di un kilometro a partire dal *Point Of Interest* “**The Pond**”, basterà chiamare l'apposita funzione **7** e passare i dati “**The Pond**” e 1000. Verranno elencati (se disponibili) i *Point Of Interest* richiesti;
 - Per visualizzare le strade percorribili a partire da un dato *Point Of Interest* bisognerà chiamare la funzione **9** e passare il nome del nodo in questione, ad esempio “**Zoo School**”;
 - Una volta terminate le operazioni necessarie, bisognerà inserire il valore **0** per chiudere correttamente il programma.