Graph Visualization of the Environmental Backcloth

Finian Lugtigheid

July 25, 2023

Introduction

This project provides a system to create graph models from multiple datasets with a focus on datasets that have location information. The system is demonstrated by creating a graph model of the city of Vancouver, Canada and analyzes the model using a metric called *reach* which has been adapted from social network analysis.

This is an extension of the work done in 2022 by Sheshadri et al. Their work is availiable on github at https://github.com/aadithya-sesh68/mitacs-cacm-project-2022 and was published in the paper "Graph Model of Environmental Backcloth" https://ieeexplore.ieee.org/abstract/document/9946572

The github page for this project is located at https://github.com/Slow-Swift/gveb-2023

There are three main components to the system:

- data_wrangler, a set of python scripts for analyzing, cleaning, and loading data
- Neo4j, an online graph database
- GraphXR, a website for visualizing large graphs

Project Setup

You should already have python and git installed and know how to use both.

Begin by cloning the project github reposity: https://github.com/Slow-Swift/gveb-2023.

Create a virtual environment by running `python -m venv env_name` and activate it by running `source env_name/bin/activate` on Linux or Mac or `env_name\Scripts\activate`.

Install the required python packages by running `pip install -r requirements.txt`.

Create a Neo4j account and create a database. Each Neo4j account can create one free database. When creating the database you will be prompted to download a file containing the database connection information. Place that file in a known folder (the directory containing the project's root folder is recommended) and change `DATABASE_INFO_FILEPATH` in data_loading/main.py to point to that file.

Create a GraphXR account. Then add a project and select configure Neo4j Instance. For Neo4j Host enter the Neo4j URI in the database connection info file. Only enter the portion of the uri following '//' i.e. don't enter 'neo4j+s://'. Leave the Bolt field as is and verify that Database Name and Neo4j Username are correct. Enter the Neo4j Password and click confirm.

Open the GraphXR Project, click on the project pane (file icon) on the left side of the window and choose the Extensions tab. Select Grove from the dropdown list. Click the '. . . ' button, choose import files and load all the .grove files from grove_notebooks.

(Optional) If you want to use the street view you will need to create a Google Maps Javascript API key. To do so you will need to create a Google developers account. Using it for street view is unlikely to ever cost you money as you get $200 of free credit each month and in addition I never found that it was ever using any of that free credit. Once the API key has been created click the '. . . ' button on GraphXR's grove, choose settings and switch to the Secrets tab. Create a new secret called 'Google API key' and set the value to the newly created API key.

To upload the graph model to Neo4j run `python main.py` from the data_loading folder. The Graph Management grove script can then be used to pull the graph into GraphXR.

Using GraphXR

GraphXR allows massive graphs, such as the entire street network of Vancouver to be visualized. It has functionality to do some pretty neat things such as using node latitude and longitude to place the graph on a map. Unfortunately it is not well documented and it can be quite hard to find out how to use it properly. A couple starting points are the GraphXR User Guide https://static1.squarespace.com/static/5c58b86e8dfc8c2d0d700050/t/5df2bc684c2e38505cf2be1c/1576189042217/GraphXR_User_Guide_v2_2_1.pdf and the How To GraphXR powerpoint series https://helpcenter.kineviz.com/learning-center/HC/. The main thing to know is that clicking the world icon on the left hand side of the window causes GraphXR to display the graph on a map.

To aid in loading the model from Neo4j without having to write complicated Cypher queries, the Graph Management Grove script was developed. This renders a web page with a map and various buttons. A region can be selected on the map using left click and shift + left click and node categories can be loaded or unloaded using the buttons.

To use the street view, select a node and click the Street View button on the grove script.

Grove is nice because it can allow greater analysis of the graph model but it is even harder to work with the GraphXR itself. The documentation is extremely lacking and in some places is just wrong. In addition the functionality that the API gives you is quite limited in some areas. For what its worth, the API documentation can be found at https://kineviz.github.io/graphxr-api-docs/. You can also check out the graph_management and street_view grove scripts to see examples of how it can be used (check them out on GraphXR's grove not as a file). In your grove scripts I recommend specifying version 0.0.227 while getting the API, eg `api = getApi("0.0.227")`, as this matches the documentation the best.

Neo4j

Neo4j is an online graph database service. The free database each account can use can have up to 200,000 nodes and 400,000 relationships. The graph model of Vancouver uses about 40% of the node limit and 21% of the relationship limit so the free account should be good enough for most purposes. For the moment, Neo4j is mostly used as an intermediary between the python scripts and GraphXR. Despite that, it has quite a robust query language called Cypher which be used to manipulate the graph model. In the future more research should be done into how we can take advantage of Neo4j and Cypher to better analyze the graph model. The Cypher documentation can be found at https://neo4j.com/docs/cypher-manual/current/introduction/

Python

Most of the project consists of Python scripts which deal with cleaning up data, loading it to Neo4j, and analyzing it.

The important folders of the project are structured as follows:

- GVEB-2023
    - data
        - cleaned_data
        - original_data
    - data_cleanup
    - data_loading
    - data_wrangler
    - gui
    - reach_visualization

Data Wrangler

The data_wrangler folder is the most important in the entire project. It is essentially a package that provides functionality for managing the datasets, cleaning them up, creating node categories and relationships between nodes, and loading everything up to Neo4j.

Although this folder is like a package, I have neither uploaded it to PyPi or installed it on my system. In order to access it like a package, each script that uses it currently adds the GVEB-2023 folder to module seach path. This is done through the system package:

```
import sys
sys.path.append('../')
```

Dataset

The heart of the data_wrangler package is dataset.py and the Dataset class. The Dataset class loads data from a csv file and provides functionality to modify the data by renaming, deleting, or adding columns. The datatype of a function can be set by applying a function to each row of the column.

To load a file use `Dataset.load_file(file_name, conversion_map=None, delimiter=',')`

This method is quite flexible allowing a dataset to be almost completely cleaned up by using the conversion map. The conversion map is a dictionary with column names of the resulting dataset as keys. Values should be:

1. A function that takes in a single string parameter. eg. int
2. A tuple of two elements. Element 1 being a function as described above. Element 2 should be a fieldname from the csv file. ()
3. A RowFunction

Option 1. is used to set the type of a column. Option 2. is used to set the type of a column and name is something different than the csv file fieldname. Option 3. is used when more than one column is needed to determine the value of a column.

An example conversion function is

```
{
    "id": int,
    "name": (str, "Name"),
    "indexName": RowFunction(lambda row, i: str(i) + str(row["Name"]))
}
```

Despite the functionality provided by conversion map I no longer recommend using it. Instead load the dataset without a conversion map and use the functions `dataset.rename` and `dataset.convert_property` because it is much more obvious what these do. If you need to apply a RowFunction use `dataset.add_property`.

Dataset provides much more functionality for manipulating datasets, including removing columns, taking the cross product of two datasets, combining two datasets, and matching the rows of two datasets based on their latitude and longitude. The documentation for this class is unfortunately not complete at the moment but exploring scripts that use it such as data_loading/main.py and data_cleanup scripts can hopefully make understanding it easier.

Category

Category is a simple class used to represent a category of nodes in a graph. Its constructor takes in the name of the category, the dataset to use for the nodes and a list of the properties that each node should have. When writing the category to Neo4j a node is created for each row in the dataset and the properties listed are copied from the row to the node. Each element of the property list is usually just a string but can be a tuple of two strings where the first string is the name of the nodes property while the second is the name of the column in the dataset. I recommend renaming the columns in the dataset instead of using this.

Relationship

The Relationship class defines a relationship between two categories of nodes. Its constructor takes in the name of the relationships, the two categories that is goes between (can be the same category twice), the column name in the dataset of the first category that links to the primary key of the second category, and a RelationshipPropertyMatcher that selects the properties for the relationship. The property matcher can be None in which case the relationship will have no properties (more on RelationshipPropertyMatchers later). The relationship is always from the first category to the second.

Graph Writer

GraphWriter is a class that manages writing data to Neo4j. Its constructor takes in a Neo4j Connection Session. To clear all the data from the Neo4j Database call `clear_all()`. If the database is not cleared, you will likely end up with duplicate data and quickly run out of room in Neo4j. For each category that needs to be loaded call `write_category(category)` and for each relationship call `write_relation(relationship)`. Note that both categories that the relationship connects needs be written before writing the relationship.

Writing data to Neo4j will often look like the following:

```
driver = GraphDatabase.driver(uri, auth=(user, password))
with driver.session() as session:
    if not session: return
    writer = GraphWriter(session)
    for category in categories:
        writer.write_category(category)
    for relation in relationships:
        writer.write_relation(relation)
driver.close()
```

Conversion Functions

conversion_functions.py provides a few handy functions that can be passed as an element of a ConversionMap in `Dataset.load_file` or `Dataset.convert_property`. `convert_if_not_null` takes a function that will be run if the value is not null and a default value that will be returned if the value is null. This is useful when trying to set the type of a column that has null values that may cause an error. `create_regular_str` makes sure that a numberic string always is at least 2 digits long by padding it with zeros on the left. `split_latitude` and `split_longitude` get the latitude and longitude components of a string of the format `'latitude, longitude'`. RowFunction is a wrapper class that holds a function simply so typing can correctly identify it as a row function. This is a really hacky workaround and RowFunction should not be used. Please use `Dataset.add_property` instead of using RowFunctions in the ConversionMap. To get an idea of how conversion functions are used take a look at data_loading/main.py.

Relationship Property Matchers

relationship_property_matchers.py provides functions that collect properties from rows so that they can be used by relationships. `first_set_property_match` takes in a list of properties that will be pulled from the first category in the relationship. `second_set_property_match` is the same except that it pulls properties from the second category. `dual_set_property_match` takes two lists, the first list pulls properties from the first category and the second list pulls properties from the second category. To get an idea of how these are used take a look at data_loading/main.py

Data

Data conatins the dataset which are used to create the graph model. Each datafile should be a csv but can use any delimiter. Most of the data in original_data are not modified from when they were downloaded. The exceptions to this are streetsegments which were given location information and business-licences because the downloaded file was too large for github so I replaced it with the cleaned file so that scripts would still run. The original business-licences file can be downloaded from Vancouver's Open Data Portal (see below).

The data files were downloaded from:

- business-licences: https://opendata.vancouver.ca/explore/dataset/business-licences/information/?disjunctive.status&disjunctive.businesssubtype
- rapid-transit-stations: https://opendata.vancouver.ca/explore/dataset/rapid-transit-stations/information/
- schools: https://opendata.vancouver.ca/explore/dataset/schools/information/
- storefronts-inventory: https://opendata.vancouver.ca/explore/dataset/storefronts-inventory/information/
- crime: https://geodash.vpd.ca/opendata/
- junctions: https://www.esri.com/en-us/home
- streetsegments: https://www.esri.com/en-us/home
- transitstops: https://www.translink.ca/about-us/doing-businesswith-translink/app-developer-resources

I am not sure exactally where the junctions, streetsegments, and transitstops were downloaded from because they were downloaded for the previous paper.

The storefronts dataset is not used anymore and has been replaced with the business licences. This is because there were many stores missing from the storefronts dataset. The business dataset was used by filtering for business types that had retail or wholesale in the name and were valid after the year 2021.

cleaned_data contains the dataset files after they have been cleaned by data_cleanup

Data Cleanup

The data_cleanup scripts process the data files from data/original_data. The first step is renaming.py which does an initial cleaning pass that simply renames some columns, removes unnecessary columns, and combines the two crime files. The business data is handled separately by businesses_cleanup.py because it is much more complex. The business are filtered based on their business type and whether they had a valid licence after 2021. Following the initial sweep, cleanup.py finds the connection between all the different datasets by comparing the latitude and longitude. All data layers except the street network are conencted to the nearest junction and each junction receives an extra property containing the number of nodes in each layer that is connected to it. The cleanup process also removes many entries based on missing data or being more than 200m from a junction. The final step comes in reach_calculation which calculates *reach* (more details later) for each junction. The results are stored in data/cleaned_data.

compute_segment_locations.py was run before any of the others files and the results were stored in data/original_data/streetsegments.csv. This was somewhat by accident but doesn't affect anything as it didn't overwrite any data. This script finds averages the locations of both junctions that a street segment is connected to in order to find a reasonable estimate for the location of the street. This was used in the past to position street nodes but those have since been replaced with relationships and so these values are not really used.

Data Loading

The data_loading folder contains just one script, main.py which loads the datasets, created the necessary categories, and relationships and then writes everything to the Neo4j database.

Reach Visualization

The scripts inside reach_visualization display the results of reach calculations inside matplotlib. There are a few different visualization tactics including scatter plots, best fit lines, correlation matrices, and histograms. To visualize each of these I just commented out the other ones depending on what I wanted to see.

Gui

The gui folder contains the beginning of work towards creating a nice graphical user interface for cleaning up data, creating the categories and relationships and writing it to Neo4j. The goal is that any simple cleanup such as renaming and removing columns, filtering datasets and so on can be done directly in the UI. Ideally there would also be an option to run simple python scripts from the UI. Categories and Relationships will also be able to be created and everything will be saved so that the cleanup can be rerun or modified without hassle. Advanced analysis like reach may still need to be done outside of the UI by writing python scripts.

The gui is currently made using tkinter and is only availiable on the gui git branch. Currently the user can load data files, rename and remove columns, and set the primary key of the dataset. In addition the user can create categories using the loaded datasets. The primary key of the dataset is always used as a property and the user can select additional properties. More work needs to be put into working out what happens when columns are renamed or removed from a dataset after they have been added as properties in a category.

Most of the changes a user makes are saved and can be recreated when the program is loaded again. This was accomplished by defining actions which have a text representation similar to terminal commands. These actions are created as the user makes changes and are saved to a file when the user closes the program. They changes can be recreated by parsing the action file, creating the actions, and executing them.

Reach

Reach is a centrality measure used in Social Network Analysis. I have adapted it to suit the street network while keeping the main ideas of reach. The formula I am using is

$$R_{ji} = \sum_{k \in J} A_{ki} \cdot e^{-\frac{1}{2}\left(\frac{d_{jk}}{\sigma}\right)^2}$$

Where R_{ji} is the reach of junction j for attribute i, J is the set of all junctions, A_{ki} is attribute i of junction k and d_{jk} is the distance between junction j and junction k. For a junction j this formula essentially finds the distances between j and all other junctions, passes those distances through a normal function and weights those distance by attribute i of each of the junctions. This is done for each of the junctions. For more details on why this is the formula I'm using, check out the paper (not published yet).

The biggest challenge with this calculation is calculating the distance between all pairs of junctions. Although Floyd-Warshall's algorithm accomplishes this, running Dijkstra's algorithm for each junction actually turns out to be better because the street network is not a very dense graph. Dijkstra's also has the added benefit that it is easy to limit how far out it pathfinds which can speed up calculations by not incorporating distant junctions which will not change the value much.

Troubleshooting

If the connection to Neo4j is continuously failing, check to see that the database is running. If the database has not been accessed for a while (a few days) then it stops running.

Due to internet connection issues, loading data to Neo4j sometimes fails. Try running the loading script again.