Similarity Search in Heterogeneous Networks

Udhav Sethi
Indian Institute of Technology Hyderabad

Harmanpreet Singh
Indian Institute of Technology Hyderabad

1. Abstract

Similarity search is a basic operation in a database. Most of the existing similarity measures are defined for homogeneous networks. However, most of the information networks today are heterogeneous, with different types of objects connected through different relationships. In this project, we aim to implement a similarity search in a heterogeneous graph.

For the purpose of this project, we used a subset of the IMDB database as our dataset. We divided the project into two phases. Phase 1 includes data extraction and filtering, and the organization of data in Neo4j, a graph-based database. Phase 2 includes defining the similarity measure for two nodes using the PathSim algorithm and developing a REST API for similarity search and a user interface for query making.

2. Introduction

Graphs are mathematical structures used to model pairwise relations between objects. Graphs can be used to model many types of relations and processes in physical, social, and information systems, as many practical problems, can be represented by graphs.

The modern-day graphs used in real systems are often heterogeneous, i.e., they contain different types of nodes, representing different entities, and different types of edges interconnecting these nodes, representing different relationships. It is important to provide an effective search solution in such networks, as it can be used in various applications, such as recommendation systems.

The heterogeneous graph we use in this project is the IMDB graph, consisting of the node types movies, directors, genres, actors and actresses. The relationships of these nodes are defined as acted_in, directed_by, etc. The aim of the project is, given a node in the graph, find other nodes from the graph which are nearest to the given node, based on the defined distance metric.

3. Implementation

Phase 1: Data Extraction, Filtering, and Organisation

A. Data Extraction

For this project, we used a subset of the IMDB database, which was provided for use at http://www.imdb.com/interfaces. The data was downloaded from the FTP server at http://ftp.fu-berlin.de/pub/misc/movies/database/ in the form of list files containing unstructured information about actors, directors, genres, movies. plots, cinematographers, etc. The list files for only actors, actresses, directors, movies, movie ratings, and genres were extracted, containing the following data format:

Actors:
(Lastname, Firstname #TITLE <!(detail)/> <!(detail)/> <![role]/> <!<billingPosition>/>)
Actresses:
(Lastname, Firstname #TITLE <!(detail)/> <!(detail)/> <![role]/> <!<billingPosition>/>)
Directors: *(Lastname, Firstname #TITLE <!(detail)/>)*
Ratings: *(Dist.Num Votes Rank **#TITLE**)*
Movies: *(**#TITLE** Year)*
Genres: *(**#TITLE** Genre)*

Legend:
<!xxx/> : *optional*
#TITLE :
name (year) <!(info)/> <!{<!episodeName/><!{episodeNum}/>}/> <!{{SUSPENDED}}/>

B. Data Filtering

The next step was to structure and filter the data properly as per the requirements of the project. The list files were used to generate structured CSV files for each of the entities. Each CSV file was obtained after multiple levels of filters were applied using python scripts. The filters applied, and the data statistics obtained after each step are described below.

1. Converting list files to CSV files

The list files were iterated and useful information was extracted to structure the data into CSV files as follows:
Movies: *(title,year,rating,votes)*
Actors: *(first_name,last_name,movie_title)*
Actresses: *(first_name,last_name,movie_title)*
Directors: *(first_name,last_name,movie_title)*
Genres: *(movie_title,genre)*

2. Loading CSV files into the database

Challenge faced: Large file sizes
The output CSV files were very large in size and couldn't be loaded into the Neo4j database using the web interface. Each trial resulted in a database disconnection error because too much time went into loading these large files, leading to a timeout or memory error in most cases.

Approach 1: Splitting the data
We tried to split the data into multiple small files and tried to load each file individually. However, the problem still remained, and the size of the file successfully loading into the database was too small.

Approach 2: Data Filtering using SQL
The data needed to be filtered in order to insert into the database so as to use it for phase 2. To collect the stats for the data and filter accordingly. We tried to load subsets of the data into a MySQL database. But again due to large file sizes, we failed to do so.

Approach 3: Manual Filtering
Finally, we decided to manually filter the data by deciding some factors and thresholds and wrote scripts in Python to obtain the desired filtered data sets.
The data was progressively filtered on the following levels**.** The scripts used for filtering are mentioned in parenthesis followed by the stats after each step:

Initial Stats:
#Movies: 3439882
#Actors: 15643508
#Actresses: 9395847
#Directors:2273151

A. **Removal of movies without rating data (***tsv2csv.py***)**
   Stats after filtering:
   #Movies: 645242

B. **Removal of TV series and video-games by removal of tags (TV), (VG), (V), {}**
   (*filter_VG_TV.py,filter_V.py,filter_bracket.py*)
   #Movies: 321534
   #Actors: 4885618
   #Actresses: 2443331

C. **Removal of movies released before 2000** (*filter_year_2000.py*)
   #Movies: 158062

D. **Removal of movies with a number of votes<100** (*filter_number_of_votes.py*)
   #Movies: 39735

E. **Finding distinct #Actors and #Actresses for filtered movies**
   (*distinct_actors.py, distinct_actresses.py*)
   #Actors: 320797
   #Actresses: 180271

F. **Removal of actors and actresses who acted in only one movie released before 2010** (*distinct_actors_2010.py, distinct_actresses_2010.py*)
   #Actors: 86103
   #Actresses: 47143

G. **Removal of movies without any associated actors or actresses**
   (*find_union_movies.py*)
   #Movies (with associated actors): 32120
   #Movies (with associated actresses): 32056
   #Movies (union of both): 32770

H. **Filtering of directors for resultant movies** (*filter_directors.py*)
   #Directors (with associated movies): 36248

Challenge: Large computation time

In many of the above filtering operations, we needed to compare two files containing a large number of lines. With two such files containing *m and n* tuples each, the iterative comparison through brute force leads to a time complexity of **O(m*n)**. Since *m and n* were very large, this was highly inefficient, and the system was unable to handle the computation.

Approach: Increased efficiency using Hashmaps and IO buffers
To overcome the above challenge, we used alternative approaches. We iterated over the file containing m tuples and stored the *m* values in hashmaps. After that, the *n* tuples in the other file were iterated upon and matched with the values stored in the output. This reduced the problem to **O(m+n),** greatly increasing efficiency.
Another approach we chose is to store the input and output buffers instead of reading from and writing to files on the fly. This reduced intermittent IO time and helped increase efficiency.

Result
After applying the above progressive filters, we obtained the following stats:
#Movies: 32770
#Actors: 86103
#Actresses: 47143
#Directors: 36248

C. Data Organisation

Finally, after filtering the data and structuring it into CSV files, the data was entered into a graph database, so as to visualize the entities and their relationships. The **Neo4j database** was used for this purpose. Data were entered with the help of the **Cypher** query language and the **Py2neo** client library in the following steps:

1. The movies.csv file was simply imported into the Neo4j database, creating a node per movie.
2. To import actors into the database, py2neo scripts were written so as to check for each actor-movie pair, whether the node (the entity) for the actor already exists in the database. If so, the existing node for the actor was linked to the paired movie with the help of an edge (the relationship). Else, a new node was formed for the actor and a relationship was formed with the existing paired movie.
3. Step-2 was repeated for actresses, directors, and movie genres.

As a result of the above steps, we obtained a heterogeneous graph representation of the filtered data in the Neo4j database, complete with nodes representing the movies, actors, actresses, directors, and movie genres, and edges representing their corresponding relationships.

Phase 2: Similarity Search in a Heterogeneous Network

Heterogeneous information networks are the logical networks involving multiple typed objects and multiple typed links denoting different relations. Similarity score in such networks is computed considering different linkage paths in the network. In this way, one could derive various similarity semantics for the network.

The concept of meta path-based similarity is used to compute the similarity scores between nodes using predefined paths consisting of a sequence of relations defined between different object types. The similarity measure called PathSim is able to find peer objects in the network, which turns out to be more meaningful in many scenarios compared with random-walk based similarity measures.

Under the proposed meta path-based similarity framework, there are multiple ways to define a similarity measure between two objects, based on concrete paths following a given meta path. One may adopt some existing similarity measures, like SimRank, P-PageRank etc. However, these measures are biased to either highly visible objects (i.e., objects associated with a large number of paths) or highly concentrated objects (i.e., objects with a large percentage of paths going to a small set of objects). But the new similarity measure PathSim is able to capture the subtle semantics of similarity among peer objects in a network. PathSim can identify objects that not only are strongly connected but also share similar visibility in the network given the meta path.

Algorithm: PathSim

PathSim is a new meta-path based similarity measure framework that captures the subtlety of peer similarity in an information network. PathSim is confined to round-trip symmetric meta-paths as peer relationships should be symmetric.
Given a symmetric meta path P, PathSim between two objects of the same type x and y is:

$$s(x,y) = \frac{2 \times |p_{xy}|}{|p_{xx}| + |p_{yy}|}$$

where $\mathbf{p_{xy}}$ *is a path instance between* ***x*** *and* ***y***, $\mathbf{p_{xx}}$ *is that between* ***x*** *and* ***x***, *and* $\mathbf{p_{yy}}$ *is that between* ***y*** *and* ***y, all of the form*** ***P.***

Thus, given a meta path P, *s(x, y)* is defined for x and y in terms of two parts:

1. Their connectivity, defined by the number of paths between them following P
2. The balance of their visibility

Properties of Pathsim

1. Symmetric: $s(x_i, x_j) = s(x_j, x_i)$
2. Self-maximum: $s(x_i, x_j) \in [0, 1],\ and\ s(x_i, x_i) = 1$
3. Balance of Visibility: $s(x_i, x_j) = 2 / (\sqrt{M_{ii} / M_{jj}} + \sqrt{M_{jj} / M_{ii}})$

Meta paths implemented

Our similarity computation engine supports finding similar nodes for the following node labels, based on the mentioned meta paths. The python files used for implementation are of the same name as the meta-paths.

Actors:

3 length meta paths:

AMA

5 length meta paths:

AMAMA
AMAxMA
AMDMA
AMGMA

Actresses:

3 length meta paths:

AxMAx

5 length meta paths:

AxMAMAx
AxMAxMAx
AxMDMAx
AxMGMAx

Directors:

3 length meta paths:

DMD

5 length meta paths:

DMAMD
DMAxMD
DMDMD
DMGMD

Movies:

3 length meta paths:

MGM

Weighted meta paths:

Weighted combination of MGM, MAM, MAxM, MDM

where:

A: Actor
Ax: Actress
D: Director
M: Movie
G: Genre

4. Attempted modifications to PathSim Algorithm:

1. Using Jaccard Index in 3 length meta paths

The similarity between two nodes according to the PathSim algorithm is defined as:

$$s(x,y) = \frac{2 \times |p_{xy}|}{|p_{xx}| + |p_{yy}|}$$

where ***p***

*where $\mathbf{p_{xy}}$ is a path instance between **x** and **y**, $\mathbf{p_{xx}}$ is that between **x** and **x**, and $\mathbf{p_{yy}}$ is that between **y** and **y**, all of the form **P**.*

Considering meta paths of length 3, the similarity between two nodes is analogous to:

$$s(x,y) = 2 * \frac{|A \cap B|}{|A| + |B|}$$

This term is similar to the Jaccard similarity coefficient:

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

So, we tried a modification of the length 3 meta-path algorithm by replacing it with the Jaccard Index and defining similarity as:

$$s(x,y) = 2 * \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

2. Finding similarity using multiple weighted meta paths

In the PathSim algorithm method, the similarity between two same types of nodes in a heterogeneous graph is defined based on a symmetrical meta-path defined between them. Now, if we need to incorporate multiple nodes and relationship types into the similarity index, we would have to define a fairly long meta-path, which will also need to be symmetrical. This would make the index less accurate, as well as extremely computationally extensive to compute.

For example, in the IMDB graph, the 'movie' label (M) nodes have relationships with genres (G), actors (A), actresses (Ax), and directors (D). Now, to incorporate all these into a single similarity measure, one of the shortest symmetrical meta paths we would need to define is:

M-A-M-Ax-M-D-M-G-M-D-M-Ax-M-A-M

This approach has several problems:

1. The meta-path is too long to accurately find 'strongly connected' movies.
2. We cannot assign more weight to one kind of label (say, genre) over other labels explicitly.
3. Computation of PathSim using this meta path would be fairly computationally extensive.

To deal with finding similar nodes which have direct connections with a lot of label types, we propose the following similarity index:

$$sim(x,y) = \frac{\Sigma \alpha_i \, s(x,y)}{\Sigma \alpha_i}$$

where $\alpha_i \in [0, 1]$ and s(x,y) is the similarity score of computed using meta paths of different lengths. $\alpha_i$can be tested for different values and similarity scores can be computed.

In our implementation, we included computation of similar movies based on weights assigned to all the different kinds of nodes, using only 3 length meta paths:

$$sim(m1,m2) = 0.75 \times sim_{MGM} + 0.25 \times (sim_{MAM} + sim_{MAxM} + sim_{MDM})$$

5. Technologies and Standards Used

Neo4j

To study heterogeneous networks, we decided to use a graph-based database. While other databases compute relationships expensively at query time, a graph database stores connections as first-class citizens, readily available for any “join-like” operation. Accessing these connections is an efficient, constant-time operation and allows to quickly traverse a large number of connections in a short amount of time.
Neo4j is a highly scalable graph database that is efficient in handling graphs in terms of data storage and query performance. Also, Neo4j supports *Cypher*, a native graph query language that provides an expressive way to describe relationship queries. This was useful for defining meta paths and efficiently computing the similarity index.

Using Cypher (Neo4j) to efficiently compute the PathSim similarity index
As we know, the Pathsim algorithm defines the similarity between two nodes as follows:

$$s(x,y) = \frac{2\times|p_{xy}|}{|p_{xx}|+|p_{yy}|}$$

Now, to find the top k similar nodes given a query node x, we would have to iterate over all the nodes of the same label as *x* and compute the similarity.
But using Neo4j, we are able to shortlist the nodes to be considered for top k results.
For example, to find similar actors for ‘Brad Pitt’ using the meta path AMA, we follow the following approach:

1. First, we find all the actor nodes which are reachable from node x (‘Brad Pitt’) through a pattern AMA, and label them as y:
2. Then, we compute p_{xx} , p_{xy} and p_{yy} considering only these ‘y’ nodes.
3. Using these p_{xx} , p_{xy} and p_{yy} values, we compute the similarities and sort them to find the top k results.

Thus, our computation time decreases as the nodes not reachable from x via the defined meta paths did not have to be considered.

Py2neo

Py2neo is a client library and toolkit for working with Neo4j from within Python applications and from the command line. Using Py2neo we were able to develop scripts to run the Cypher queries one by one and store their results in a Hashmap, which allowed us to execute the computations in a space-time efficient manner.

Django

Django is a high-level Python Web framework. Using Django, we were able to develop and deploy a RESTful API to serve the requests made by our front-end web application, by executing the Py2neo scripts and storing the results as JSON files.

REST API

The similarity computation engine hosted on the Django server uses the REST (Representational State Transfer) architectural style. This enables stateless transfer, i.e., the client-server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and the session state is held in the client. This enables the service to be used by any client communicating appropriately with the API using HTTP request methods like GET, POST etc.

HTML, CSS, JS

Standard web technologies HTML, CSS, and JS were used for the web application, serving as the front end to make queries on the Django server.

6. Conclusion and Future Work

In this project, we worked on the extraction and filtering of the IMDB data, and its organization into the Neo4j database. Also, we implemented a similarity computation engine as a RESTful API based on the PathSim algorithm.

We learned that the choice of meta-paths is subjective to the data being handled and thus is left to the user in our implementation. During our experiments, we learned that very short meta paths are not a very accurate measure for finding node similarity, However, the convergence of PathSim algorithm with respect to path length is usually very fast and the length of 10 can almost achieve the effect of a meta-path with an infinite length.

Future work may include improving performance by using indexing or other methods as the present implementation gets computationally intensive for longer meta path computations. Also, finding the best meta path or combination of weighted meta-paths for the IMDB dataset through machine learning is another possible extension.