Biolink: Interactive Exploration of Connected Biological Knowledge

Raghavprasanna Rajagopalan

December 2019

1 Abstract

What compounds have been tested in bioassays (having certain side effects) that have associations with (bind, upregulate, downregulate) genes that have correlations with the disease Asthma?

Our hope is that by providing a framework where users can ask these questions and get answers backed by research, future research can proceed in a more focused way. Traditionally relational database models have been used to store the vast amount of data that exist pertaining to biology. Databases like Disgenet and the Gene Ontology Functional Annotations have helped make biological knowledge much more accessible to other research efforts. There has been some extensive work leading efforts to connect this data, but these efforts weren't built with extensibility in mind. We provide the foundations for what a configurable/extensible framework for building and exploring a database of connected knowledge might look like. We leverage the graph database model and Neo4j to construct the network. We expose this data via a query-able REST API, as well an informative view that users can interact with easily without necessarily having to know the Cypher query language. Finally, we aim to demonstrate that our work is not domain-specific, and can be applied to build a knowledge network from various data sources in potentially *any* subject.

2 Introduction: Task

Our goal is to provide a framework for building and exploring a database of interconnected knowledge. Previous efforts to build such a network of connected knowledge include an early version of Biolink developed by Rachlin (2011) and the Hetionet database developed by Himmelstein et al. (2017). Rachlin (2011) employed a fully relational model for connecting the data from different databases (e.g. NCBI Gene Association Database), and also leveraged an in-house transformer, Transom, to construct and handle the mapping between different identifiers referring to the same entity across multiple databases. Their work provided a web client that allowed users to navigate the data and links between genes and diseases. Hetionet made use of the graph database model to describe the relations between entities. They connected knowledge from 29 public resources to connect compounds, diseases, genes, anatomies, pathways, biological processes, molecular functions, cellular components, pharmacologic classes, side effects, and symptoms (Himmelstein et al. 2017). While their work exposes a powerful heterogeneous network in Neo4j, which in itself has great visualization capabilities, it requires end users to know how to write Cypher Queries. In addition, since the code for the project isn't publicly available, efforts to extend the network (i.e. adding new nodes/relationships) must come from the project owners.

We began our work by exploring Disgenet, which tracks the association between Genes and Diseases as these associations are supported or refuted in literature as our starting point. We integrated the data in Disgenet into Neo4j, and established a simple method to generically import data from different data sources and represent them in Neo4j. We designed some powerful queries that reflect complex questions we now have the ability to ask easily (by virtue of Cypher's syntax and the underlying graph model). We expose a REST API with endpoints for genes and diseases which automatically ask those complex questions of the data. We provide a web client that users can make use of to explore the data.

3 Process: Data & Methods

3.1 Data

3.1.1 Disgenet

Picking genes and diseases and their relation as a starting point, our first task was to explore Disgenet and becoming familiar with the data. Disgenet has its own internal gene and disease identifier that it uses when reporting associations, but also keeps track of entrezGene IDs and gene symbols (in the gene table) and disease UMLS CUI (a standardized disease ID) and the standardized disease name (in the disease table).

In addition to details about the gene like uniprotId (protein identifier for the gene's downstream protein product), Disgenet calculates certain metrics for genes, such as Disease Specificity Index and Disease Pleiotropy Index. Diseases in Disgenet belong to one of 29 MeSH disease classes. Multiple diseases can belong to the same MeSH class, and similar diseases might also be in the same MeSH class. From Disgenet documentation, the Disease Specificity Index is a measure of how many diseases the gene is associated with (with respect to the number of diseases). Disease Pleiotropy Index is a measure of the number of different disease classes a gene has associations with over the number of unique disease classes. A gene must have an association with at least one disease in a class for the class to be considered for this metric. A gene can have a high DSI but low DPI if it is associated with many diseases within a small set of disease classes. A gene can have a low DSI but high DPI if it has associations with only a few diseases in many classes. These metrics aren't yet calculated for all genes, so some null values exist in the data.

Along with disease information, disease type and disease class are also stored in two tables, joined by the internal disease id. (Disease classes and type could potentially become nodes of their own in the future.)

Most importantly, the gene-disease associations are also tracked by Disgenet. An association keeps track of which gene and disease are involved, the source/pubmed id/sentence that reported the association, as well as calculated metrics like Evidence Level, Evidence Index, and score. Evidence level is an indication of how much evidence (in the form of publications) backs the association. Evidence Index indicates that there are contradictory results in the literature. We explored the the two metrics in comparison with each other, but couldn't conclude any correlation between the two (more exploration is needed). That said, the association score seems to be a reliable metric we can use as an indicator of the likelihood that a gene is related to a disease.

3.2 Graph Database

With some understanding of the data in Disgenet, we began defining a tentative schema for the graph model. Our intuition was: an entity is a node, and a join table is a relationship (edge).

Looking at gene-disease associations, we decided that Gene and Disease serve as Node Types of our graph. Then each row in the gene table becomes a node of the Gene node type, and the same holds for diseases.

We determined that not all columns would be useful in the graph. Rather we wanted Neo4j to contain only the core information needed to connect the data, and perhaps incorporate this other data in a relational format, ultimately leading to a multi-modal database architecture. We ignored Disgenet's internal identifier for the gene, only referring to genes by the Entrezgene id for the gene, and the gene symbol. Values Z and P are 0 across all genes, so we ignored those fields as well. However, pLI, DSI, and DPI do have actual values, and espcially the latter two are calculated metrics related to diseases a gene is associated with. We keep these in our Gene node.

With diseases, we used the diseaseId (the UMLS CUI labell for the disease) as its identification. The disease name is also useful to us, along with perhaps the type. Not all diseases have a disease class associated with them, so we ignore that field for now. We could definitely consider incorporating this data at a later time.

Associations are join tables. We joined the associations table on the disease and gene tables by the internal gene and disease identifiers, and instead display entrez gene Ids and UMLS CUIs. That information is key to understanding the association. For now we track only association score in addition to the gene and disease involved, intending to use score as a measure of association strength.

We wrote scripts to SQL data from Gene, Disease, Association to CSV. Then created a cypher script to load the entities from CSV and create the Gene Node Type and all gene nodes, the Disease Node Type and all disease nodes, and then edges (:AssociatesWith) with each association score. We worked to automate this process as much as possible, and are continuing to make changes to make this data import process easier. Currently we are defining a data configuration schema in a JSON format and working on automating the data import process using that JSON. We are also looking into ETL pipelines for next steps, to manage new incoming data. We also aim to incorporate Gene Ontology functional annotations, to learn more about gene function.

3.3 REST API

3.3.1 Query Formulation

We were ready to ask questions of our database. We focused on Asthma to construct our queries. We wrote queries to find what genes are linked with Asthma, what genes are linked with Asthma that are linked to other diseases. We realized we are effectively asking "What diseases are similar to Asthma (based on the genes they are associated with)?" Essentially we are able to make assumptions about disease similarity based on their shared neighbor nodes (in this case, neighbors are genes). Essentially we're creating a bipartite graph, with genes on side and diseases on the other. We see this in Figure 1 below:

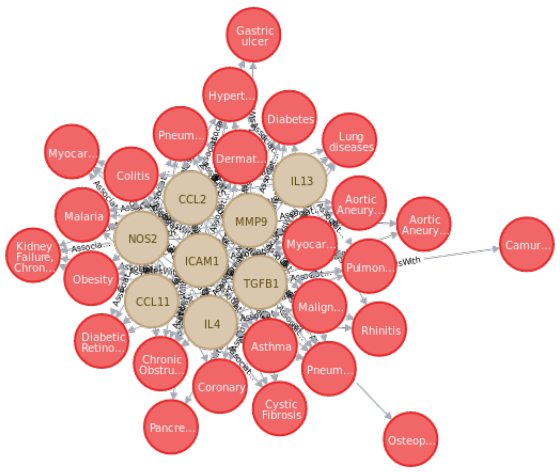

Figure 1: A bipartite representation mapping genes that Asthma is associated with that other diseases also have strong associations with. The inside (tan) nodes represent genes, and the outside (red) represent diseases. Rendered on Neo4j browser.

We worked to abstract these queries so that for a given target node and a given neighbor node type, we could return a list of other nodes of the target node type similar to it based on the associations with a shared set of neighbor nodes of the given neighbor node type. Given a gene IL4 and a neighbor node type Disease, we would return a list of genes similar to IL4 by the diseases they share in common, provided those associations lie above a threshold. Given a disease Asthma and a neighbor node type Gene, we would return a list of disease similar to Asthma by the genes they share in common with Asthma, provided those associations lie above a threshold. These queries are dynamically generated in Python3 based on an internal schema we use to map nodes and relations (and the directionality of those relations). Then when we add nodes and relations to our Neo4j instance, we need only to update the Python application's representation of the graph schema, leaving the queries untouched.

With these queries in mind, we began designing a minimal web application to support queries to our graph database, and use these query results to generate a sort of interactive view on the front end. Using Python 3 and the Flask framework, we set up a simple API server with endpoints that users can make HTTP queries to. As mentioned before, because the queries were abstracted to work for any nodes and relationships, future additions to the graph will be supported via a simple update to the schema.

3.3.2 API Endpoints Overview

An overview of the endpoints are as follows:

Path: '/gene' or '/disease'. Description: Querying this endpoint will return a list of 25 nodes of that entity type (sorted by entity name). Provide '?page=i' in the URL will allow the user to get i^{th} set of 25 entities.

Path: 'gene/geneName' or 'disease/diseaseName'. Providing the gene symbol (e.g. NOS2) or diseaseName will return a JSON containing all the information about that entity in the database. The results of the corelated neighbors database queries and similar entities queries (mentioned previously) will also be populated here.

Path: '/search'). Providing an argument to '?nodeVal=' will result in searching for all entities having properties that match the provided value. The provided argument must have a string value with more than length 3 before a search is actually run.

3.4 Client Web Application

With an API available, we began building a front-end web client to query the API and present data. Built in React, a JavaScript framework, we made use of functional React components, the Material

UI components, and asynchronous calls to fetch data from the API and render it in a comprehensible format.

We introduced a search bar as the main entry point into the app. Users can search for diseases and genes by name or ID, and as soon as they've entered more than 3 characters the search is actually run. A search with actual results will return 2 entity tables (one for the genes and one for diseases); the rows in each table will contain data for entities that match the search. Each row (item) contains a clickable link.

Upon clicking a gene link, an API call is made to dynamically render the diseases it is associated with, along with genes it might similar to based on disease associations. The same is true for disease links.

Collections (Genes and Diseases) are presented as Tables, where gene/disease names are clickable links. The front end is also built with extensibility in mind so that we are able to support future node types.

4 Next Steps

Up until now, much of our work has been setting up the infrastructure needed. We have only a small subset of the Disgenet database running in a local Neo4j instance. We have a basic API, but that too is not deployed. We have a minimal React application that queries the API, and that also is local. However, we believe that with the basic structure and design at each step of the way will make it easy for future additions and integration. There is room for improvement every step of the way.

4.1 Data & Database

We want to continue to improve the automation to build our Neo4j database dynamically from configuration files. We will focus on having support for different types of data sources. GO Annotations, functional gene annotations, are the next data source we plan to integrate. In addition, in order to make sure maintaining and updating the data happens with little to no intervention, we plan to explore building an ETL pipeline.

4.2 REST API

Currently we support similarity queries and correlation queries. However, we don't have the functionality to ask the big question that drove our work in the first place. We would like to be able to ask complex questions that involve many different entity types.

4.3 Client Side Application

Currently, search is the entry point to our data from the front end, and the results of a search are interactive tables. We want to explore creating additional views that users can switch between. A tentative view could be the equivalent of a Sankey diagram mapping genes and diseases in a bipartite graph like Figure 2 below:

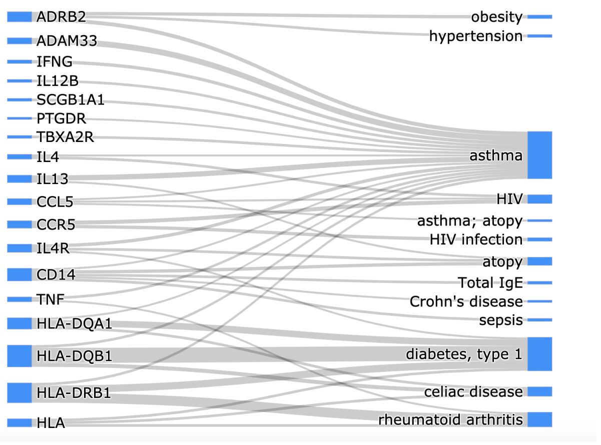

Figure 2: A bipartite graph with genes on the left and diseases on the right. The thickness of an association (line) between two entities is related to the number of publications supporting the association.

We plan on looking into the JavaScript library for Plotly to support similar visualizations on the front end. Additionally, if possible, we believe that we might be able to support filtering by clicking on entities or associations - e.g., clicking on a gene would keep only the disease associations it has and associations those genes have to other diseases (essentially what our similarity by association query returns).

5 References

1. Piñero, J., Queralt-Rosinach, N., Bravo, À., et al. DisGeNET: a discovery platform for the dynamical exploration of human diseases and their genes. Database (2015) Vol. 2015: article ID bav028; doi:10.1093/database/bav028

2. Himmelstein, D. S., Lizee, A., Hessler, C. et al. (2017). Systematic integration of biomedical knowledge prioritizes drugs for repurposing. ELife, 6. https://doi.org/10.7554/eLife.26726

3. Rachlin, J. (2011). A tool for exploring associated knowledge of biology. *Unpublished.*