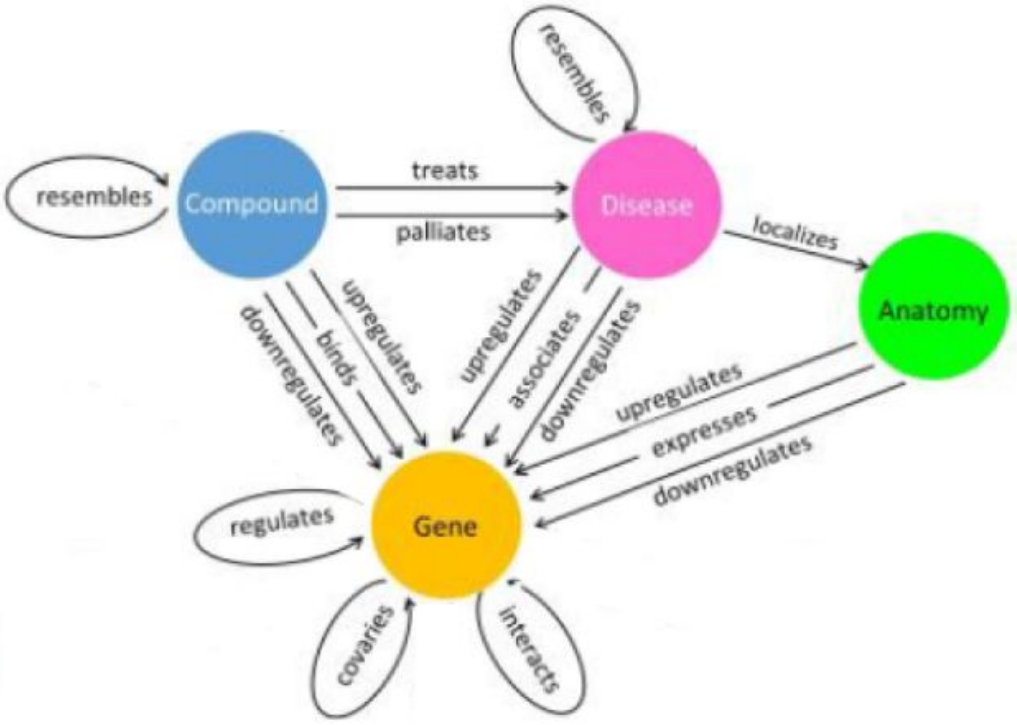
Project 1: HetioNet

Mehrab Hafiz

HetioNet is a network of diseases. It is based on nodes and edges. Nodes represent various diseases, as well as treatments, anatomy and genes related to such diseases. Edges define the type of relationship nodes.



This application uses this network to solve the following problems:

1. Find existing information about a disease and the available treatments.
2. Find all drugs that can treat a new disease.

Requirements:

- Python
- mongodb
- pymongo
- Neo4j Desktop
- Py2neo
-

Prerequisite checks:

- ./data/ contains nodes.tsv and edges.tsv files
- Mongodb service is running at default port
- Neo4j service is running at default port
- Read/Write access to data files

To start, simply open a terminal in the base project directory and run command:

python hetionet.py

Query 1: Using mongodb

- **Database Design**

  The application needs to fetch a disease with it's ID and display all relevant information. One way of achieving this is by storing diseases, genes and compounds in different related tables. However, since the data set is massive and that each query would require multiple inner joins, a relational database would be very inefficient.

  Storing each disease as a document, with all related information can reduce the query time complexity from polynomial to linear. Thus, mongodb is suitable.

  Document Model:

```
disease = {
  id: 'id',
  name: 'name',
  localizes: [],
  treatments: [],
  palliatives: [],
  genes: []
}
```

- **Queries**

  Since update operations in mongodb are more costly than find operations, update queries are avoided completely. As the application parses "nodes.tsv" file, it generates a document for each disease and stores them in a dictionary. After all the documents are

generated, it parses the “edges.tsv” file and adds items to the arrays (eg. treatment array) based on the relationship. Since these documents are contained in a dictionary, they can be looked up in O(1) time using the node id. After each disease document is updated with associated genes, treatments, palliatives and anatomy, they are added to the database one by one using the query: **db.nodesCollection.insertOne(disease)**.

Now the user can look up a disease using the disease id. Given an id, a disease is then fetched from the database using the query:

db.nodesCollection.findOne({‘id’: ‘disease_id’})

- **Improvements**

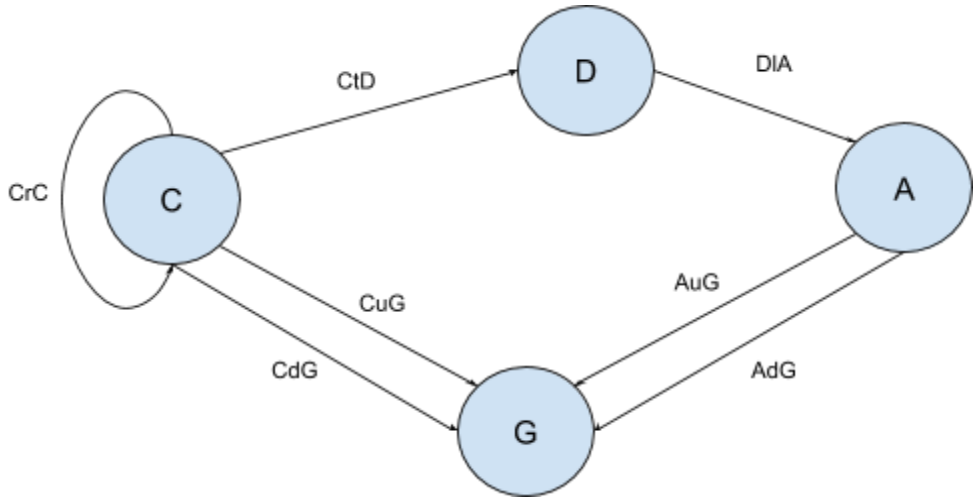
  Although the document model can yield reliable performance, it is still a linear time algorithm. In order to look up diseases in constant time, a key-value store can be used. A key-value store is faster since it uses a hash table. Each key is associated with one disease.

Query 2: Using Neo4j

- **Database Design**

  Since HetioNet is after all a graph database and this time the application needs to search for compound-disease-gene patterns in the graph, the document model just won't cut it. Not only will it take a significant amount of code to search for patterns in the database, the database will also need to keep arrays of references to genes, anatomy and treatments. In Neo4j however, complex patterns can be found using a single query.

  The difficult task here is to set up the graph from data files. The "edges.tsv" file is large and it can take hours to load up all the edges. In order to improve load times, certain relationships, that are not relevant to the query, were omitted.



- **Queries**

  Node is added to the database with:

  merge (:kind {name: “name”, iden: “id”})

  This query is run once the user selects option 2. Nodes with all its properties are added to the database.

  Edge is added to the database with:

  match (a: kind_a), (b:kind_b)

  where a.iden = "id_a" AND b.iden = "id_b"

  Only the edges ('CrC', 'DlA', 'CuG', 'CdG', 'AuG', 'AdG', 'CtD') that are needed for the drug discovery query are loaded into the database. This significantly speeds up the loading process.

  For drug discovery:

  match (c:Compound)-[:CuG]->(:Gene)<-[:AdG]-(:Anatomy)<-[:DlA]-(d:Disease {iden: "diseaseID"})

  where not (c)-[:CtD]->(d)

  match (c:Compound)-[:CdG]->(:Gene)<-[:AuG]-(:Anatomy)<-[:DlA]-(d:Disease {iden: "diseaseID"})

  where not (c)-[:CtD]->(d)

  return distinct c.name

  A single query that takes into account both CuG/AdG and CdG/AuG relationship pairs. Query result is bound to repeat, thus a distinct statement is added at the end.

- Improvements

  Nodes can be hashed to improve scalability and significantly reduce the load times. This can allow clients to look up nodes in linear time (eg. hashing node_id) instead of running match queries. Also, two separate queries can be run for CuG/AdG and CdG/AuG relationship pairs rather than one big query, speeding up the drug discovery time.