CanGraph

Release 1.0

Pablo Marcos

Dec 17, 2022

CONTENTS

CHAPTER

ONE

DOWNLOADS

There are lots of different ways you can install and use the CanGraph software:

1.1 Using a pre-built Apptainer Image

Using a pre-built Apptainer Image is the simplest, easiest way to use CanGraph: the environment is already built for you, and you can directly *start using the software*.

Note:

- To maintain confidentiality, the Public version only includes the HMDB, SMPDB and Web DataBases. You can generate the Private version yourself if you want to include more.
- It is recommended to read the *Known Issues* Section before installing

1.2 Installing the Command Line Version

You can also *install the CLI to use it on your computer*, although it will be more complicated.

CHAPTER

TWO

TUTORIALS

In these pages, you will find a list of tutorials, depicting how to use the CanGraph software and showing some useful examples that will help you understand its outputs:

2.1 Other Reading Material

There are a thousand ways that Knowledge Graphs, such as the ones *CanGraph* provides, can be explored. In this tutorial, we aim to provide a useful compilation of possible queries, ways of working, and forms of analyzing them, but this are in no way all the different ways meaningful insights can be extracted from each GraphML export *CanGraph* might provide. Neo4J is a vast software, and tons of other useful programs for graph manipulation exist, such as Cytoscape or python itself. Neo4J even provides its own data visualization tool, Neo4J Bloom, although it is not open source and might require a paid license.

We would thus like to recommend you the following, useful resources:

- Neo4J's Graph Academy hosts a free knowledge base on the field of Graphs, which can serve as useful tutorials in case what is reflected here is too confusing or difficult to follow. In particular, I would recommend:
  - Their Neo4j Fundamentals course for a primer on how graphs work and on Neo4J itself
  - Their Cypher Fundamentals course, to expand on this tutorial and on outputs processing
  - Their Building Neo4j Applications with Python course, in case you want to really dive into how CanGraph works and modify it according to your needs.
- Stack Overflow, which describes itself as "where developers learn, share & build careers" is a massive repository of questions and answers regarding all kinds of things in the informatics world. It is ikely that most of the questions you may have, or things you may want to use neo4j for, has been asked there already, so do not hesitate to search for more info or even ask there if you please.

Finally, in case you find any errors in this guides, or if you want to add any comments/suggestions, you may open an issue in our GitHub Repo. We look forward to any comments you may want to provide, and hope you have a fantastic experience using our software! :P

2.2 Installing CanGraph

CanGraph *a python utility to study and analyse cancer-associated metabolites using knowledge graphs*, presents itself in two versions: a command-line interface that enables you to run the program on any Linux-powered machine, and a pre-packaged Apptainer image.

This section of the tutorial concerns only the installation of the CanGraph **command-line interface** in a new machine. If you would rather use *one of the pre-packaged versions of the software*, you can skip directly to the *usage section*

2.2.1 Installing Dependencies

CanGraph is "a python utility to study and analyse cancer-associated metabolites using knowledge graphs"; so, of course, `python3` needs to be installed in your system; preferably, `python3.9`. A bunch of other requirements, detailed in the table below, should also be installed. Since the program is designed to be installed in a Linux system, and has been developed to run in Ubuntu, orders to install the programs using the APT package manager are provided.

Package	Description	Order to Install
python3.9	Programming language used	sudo apt install python3.9
pip	A package manager for the python package index	sudo apt install python3-pip
default-jdk	The java programming language, which Neo4j needs	sudo apt install default-jdk
default-jre	The java runtime environment, which Neo4j also needs	sudo apt install default-jre
neo4j	The neo4j DBMS, which is the backbone of the program	sudo apt install neo4j
git	A version control system, to get and manage software	sudo apt install git
curl	cURL, a C utility to download things from the internet	sudo apt install curl

All this packages can also be installed using a one-line command, which just abbreviates all the ones present above. This should be preceded by an update of the system, like so:

- `sudo apt update`
- `sudo apt install python3.9 python3-pip default-jdk default-jre neo4j git curl`
- `apt -y autoremove; apt -y clean`

2.2.2 Cloning the Git repo

Having all the required software installed, and using Git as our version control management system, we can now proceed to download the software from its Web Repository.

On any directory, open a terminal window and run:

- `git clone https://github.com/OMB-IARC/CanGraph`

The program will be now downloaded inside the "CanGraph" folder under the directory where you run git from.

2.2.3 Installing Python Requirements

CanGraph uses a lot of python packages (collections of related modules) to run in a predictable and standardized way. They are listed in the `requirements.txt` file which comes bundled with the CanGraph package; you can either install all of them manually or run the following commands from the CanGraph folder (`cd CanGraph`):

- First, update PIP: `python3 -m pip --no-cache-dir install --upgrade pip`
- And then, install: `python3 -m pip install -r requirements.txt`

2.2.4 Installing Apptainer

Note: You need to install Apptainer to use the *pre-packaged files*, but you can use the *CLI* without it.

To make the results from the CanGraph program easily reproducible and shareable, an Apptainer image is provided by us in the *downloads page*. If you want to re-generate the image yourself, either to make use of the simpler Private Image which we do not normally provide ourselves, or just to replicate our results, you will need to install Apptainer first. There are a lot of tutorials availaible online, which may or may not work on your computer, and which are sometimes split between the (old) Singularity and the (new) Apptainer brands.

To simplify the process for you, we have designed a pre-built script that installs a compatible version of Apptainer (v1.1.2) and Go (v1.43.0, the required programming language for Apptainer to run under). You can find it *here*

2.2.5 Building the Apptainer file

Once you have the Apptainer program installed and properly registered in your system (you can check this by running: `apptainer --version`), you can build the SIF image file (the one you can use to later on *run the software*) by running `apptainer build container_name definition_file`, where:

- `container_name` is the final name of the container image file; for example, `CanGraph.yaml`
- `definition_file` is the location of a file that follows the Apptainer Definition Format. In our software, we have two versions of this file, and thus two possible image files: one Private, which is built with the DataBases pre-bundled so that is is easier to use, although a bit more heavy to transmit; and one Public, in which DataBases need to be manually set up at least at first use and is thus less heavy, but more resource intensive.´

2.3 Running the Software

There are two supported ways of running the CanGraph package: you can either launch the CanGraph script from the **command line software** or you can use a pre-packaged **apptainer image**. Of these, there are two versions: a Private Image, which is built with the DataBases pre-bundled so that is is easier to use, although a bit more heavy to transmit; and a Public one, in which DataBases need to be manually set up at least at first use and is thus less heavy, but more resource intensive. Because the Private version contains copyrighted information, only the Public compiled file is provided in the *downloads page*

You can thus either download a *pre-packaged file*, compile one yourself or use the program's *Command Line Interface*; since all three options use the same underlying script (`CanGraph.main`) to work, they will all work independently of the form you choose to run the program under.

In the rest of this example, we will be providing examples on how to run the software from the command line; if you want to use an Apptainer Image instead, just replace `python3 main.py` by `./yourcontainer.sif`

2.3.1 Setting up the environment

Note: If you are running a pre-packaged file, you can skip directly to the *usage section*.

In order for the script to work properly, we need to set up some things first; the most important of them being **the DataBases folder**, which is where all important information regarding all databases will be stored. To aid in this, the *CanGraph.setup* was developed; you can read more about how it works and how to use it in its own *usage page*, but, essentially, you should run:

- `python3 setup.py -i --databases databasefolder`

where `databasefolder` is the folder where all databases will be stored (keep this value in hand because we will use it later on for the *main* module).

The software will then guide you, in an interactive an easy to understand way, through the process of setting up all the needed databases, as well as on the generation of an index that might be needed later on in the database search. Once the process is finished, you shall proceed to the next step.

2.3.2 Starting the Neo4J server

Note: If you are running a pre-packaged file, you can skip directly to the *usage section*.

As a graph-oriented software, CanGraph uses Neo4J as its underlying DataBase Management System. This means that, in order for the program to work, it needs to stablish a connection to a Neo4J database, where it will store all the relevant information. To simplify the process of downloading, configuring, and starting the Neo4J server, some functions have been created for you on the *setup* module. You can call them using the script below, which will create a neo4j folder in CanGraph's WorkDir:

- `python3 setup.py --neo4j neo4j --neo4j_username neo4j --neo4j_password neo4j`

For it to work propperly, please ensure that there are no neo4j sessions running before the command call. The program will then be installed alongside the main program, with its password being stored in a file called `.neo4jpassword`. You can then start your very first neo4j server by running: `./neo4j/bin/neo4j start`; its username will be `neo4j`, and its password can be seen by running: `head -n 1 .neo4jpassword)`.

You can alternatively start your own server, either by using a previously installed version of neo4j or by running an online version of the database; the only difference being, that you will need to provide the username, bolt adress and password to main.py as arguments.

2.3.3 Understanding the arguments

The program presents the following arguments, which can be passed, as the text below explains, just after the name of the program:

```
usage: python3 main.py [-h] [-c] [-n] [-s] [-w] [-i] --query QUERY
                       [--dbfolder DBFOLDER] [--results RESULTS]
                       [--adress ADRESS] [--username USERNAME]
                       [--password PASSWORD]
```

Named Arguments

-c, --check_args	Checks if the rest of the arguments are OK, then exits
-n, --noindex	Runs the program checking each file one-by-one, instead of using a JSON index
-s, --similarity	Deactivates the import of information based on Structural Similarity.This might dramatically increase processing time; default is True.
-w, --webdbs	Activates import of information based on web databases.This might dramatically increase processing time; default is True.
-i, --interactive	tells the script if it wants interaction from the user and more information shown to them; similar to –verbose
--query	The location of the CSV file in which the program will search for metabolites
--dbfolder	The folder indicated to `setup.py` as the one where your databases will be stored; default is `./DataBases` Default: DataBases
--results	The folder where the resulting GraphML exports will be stored; default is `./Results` Default: "Results"
--adress	the URL of the database, in neo4j:// or bolt:// format Default: bolt://localhost:7687
--username	the username of the neo4j database in use Default: "neo4j"
--password	the password for the neo4j database in use. NOTE: Since passed through bash, you may need to escape some chars Default: "neo4j"

The Query File

Of these arguments, only `--query` is required, since the program requires for at least one query to search among all the different databases. This query file should take the form of a table of Comma Separated Values, where the columns are as follows:

Name	InChI	HMDB_ID	ChEBI_ID	MeSH_ID

Inside each **comma-separated column**, you can either have one string of text (one *field*) or a series of *fields* **separated by semi-colons**. For example, you can either have:

```
Name,InChI,HMDB_ID,ChEBI,MeSH_ID
1-Amino-2-propanol,,HMDB0012136,CHEBI:19030,
```

where there is just one **Name**, **1-Amino-2-propanol**, in the name column; or:

```
Name,InChI,HMDB_ID,ChEBI,MeSH_ID
1-Amino-2-propanol;1-aminopropanol,,HMDB0012136,CHEBI:19030,
```

where **1-Amino-2-propanol** is specified as a synonym / another query of interest for the **Name** field.

This query file can be either created in LibreOffice Calc / Microsoft Excel, simply generating a new file with *as many of the proposed columns as you'd like*, and filling them with the values you want the program to search based on. Any other columns present in the CSV file will be ignored by the program, which will identify each row as the information pertaining to *a given metabolite*, thus having **as many metabolites as rows are present on the CSV, with each metabolite having as many identifiers as columns are present and as many synonyms for each identifier as semicolon values are in a given cell**.

Note: Pay special attention to the formatting of the columns; if they are wrongly specified (for example, *ChEBI* instead of *ChEBI_ID*), the program will ignore them, too.

Other Aguments of Interest

Some other arguments that require consideration are:

- **–webdbs**: When set to false (`--webdbs False`), no web databases are queried. This makes the program run fully offline, gets more reliable results and runs faster, but it returns less information.
- **–similarity**: Disables search based on similarity criteria computed using RDKit. This reduces search time, but includes less results
- **–noindex**: Runs the program by using direct search, instead of using the database index generated with the `setup` module. **The use of this option is discouraged**, as it is slower and produces less accurate results; it is presented here just for backwards compatibility and because the "Name" chain can be used as a *wildcard* field for exact text matches in the unindexed files.

Once you have the query ready, you can run it using: `python3 main.py --query inputfile.csv --password $(head -n 1 .neo4jpassword)`. This assumes that all the defaults are set to what is specified above in this same section, and sets neo4j's password to be whatever is stored in `.neo4jpassword`; you can add any additional arguments as you wish.

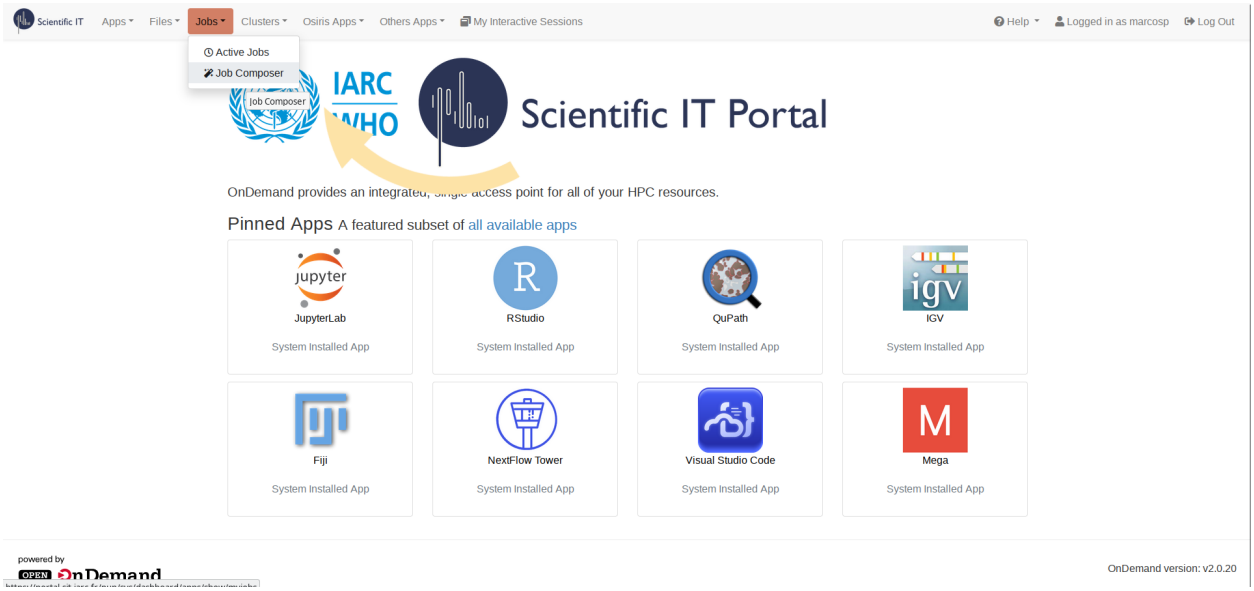
The time the program takes to process is variable, and depends on the number of matches; it may take anything from a few seconds to more than a day, as every match means, approximately, 3-5 minutes of extra processing time.

2.3.4 Running on Slurn

Note: You need to install Apptainer to use the *pre-packaged files*, but you can use the *CLI* without it.

For IARC users, as well as for any other user with access to it, the use of a High-Performance Computing platform is advised, since Neo4J is quite unstable and resource-hungry, and the program may take a long time to process during which, if using it in your own machine, it is advisable not to run any other programs at the same time. Thus, a tutorial follows:

0. **Log into IARC's HPC Portal.** There, click on *Jobs* > *Job Composer* in the top menu

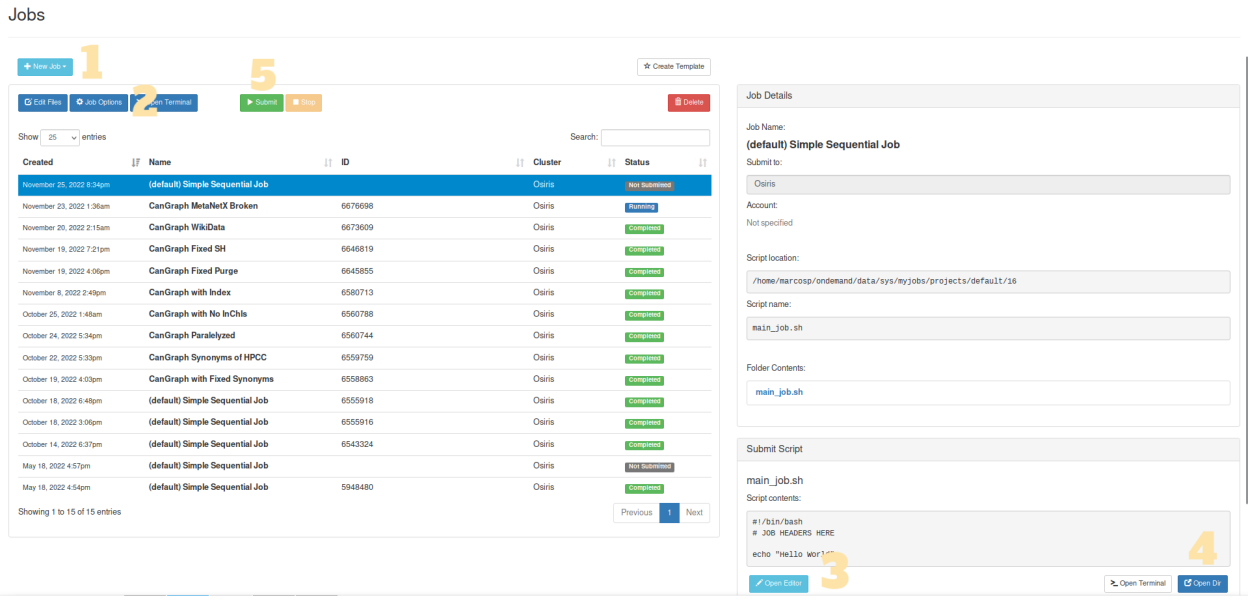


1. **Create a new Job.** To do this, click on the light blue *New Job* button, and then select one of the option; it is recommended that you choose *From Default Template*
2. **(Optional) Change the Job's name.** You can do so by selecting any job and clicking *Job Options*
3. **Modify the Runscript.** After clicking on the *Open Editor* button, an editor will appear in a new window. You should modify `main_job.sh` for it to include an order calling the script preceded by another giving the HPC permission to use it (`chmod`). For example, if you are using the pre-packaged CanGraph-Private image:

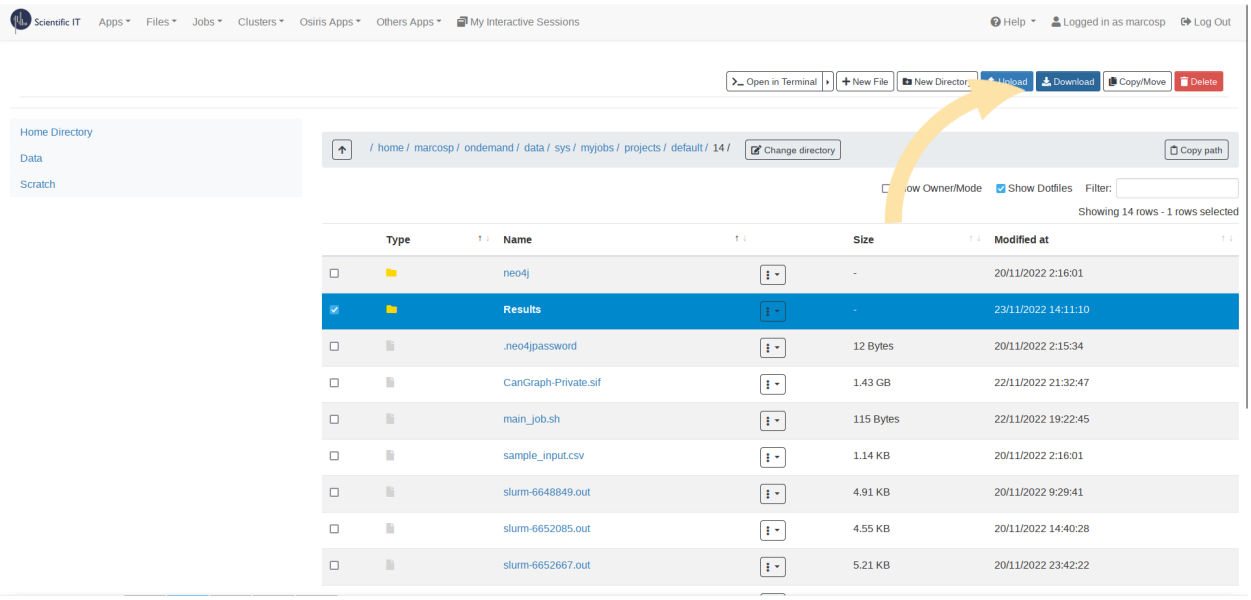
   ```
   chmod +755 CanGraph-Private.sif
   ./CanGraph-Private.sif  --query sample_input.csv
   ```

   If you are, however, using the Command Line Interface, you may want to also include the `--databases`parameter, and so on
4. **Add all the necessary files:** Click on *Open Dir*, which will open a new window containing all the files in the current WorkDir. There, you should Drag and Drop the necessary files: the `DataBases` folder in case you are using it, the different scripts and modules that form the Command Line Program (possibly in compressed `.zip` form), the apptainer image, the input file…
5. **Once everything is ready, run the file!:** You can click on the green *Submit* button, which will mark your Job in blue as *Running*. While the HPC is working on your query, a new file, called *slurm-xxxxxxx.out*, will appear on the side *Folder Contents* panel, which can also be consulted using the *Open Dir* button. This file contains an output of the different logs the program is providing, enabling you to track the process and see how it is going. Once processing is finished, a green *Completed*, or a red *Failed*, label will appear. The results will appear in the folder provided to the program as `--results`; by default, `./Results`.

Note: Sometimes, the software might not run in one-go, but might need two activations: one to install neo4j, and one for the run itself. Out of precaution, it is recommended you refresh 1 minute after submitting the job to see if it is prematurely marked as completed (with no results in the folder). If so, just re-submit the job: things should work fine this time.



6. **Download the Results:** You will find them inside the `--results` folder, and you can either download the full folder (*Select* > *Download*) or get individual files by selecting them inside the folder.



2.3.5 Known Issues

Here are some known issues you may encounter when running the software, together with possible solutions:

- Sometimes, the Apptainer-packaged version refuses to run all-in-one if it needs to set up the database folder or the neo4j program from scratch. If this happens, try re-launching the program; on this second run, it should work without problems
- On IARC's HPC system, the Public version of the Apptainer-packaged program does not work, outputting a `subprocess.CalledProcessError` with error code: `Command '['cypher-shell', '-d', 'system', '-u', 'neo4j', '-p', 'neo4j', "ALTER CURRENT USER SET PASSWORD FROM 'X' TO 'Y'"]' returned non-zero exit status 1.`. To our knowledge, this happens only on the HPC, and nowhere else; if you encounter this error, consider either building/using the Private version, or *installing the CLI*

2.4 Understanding its Outputs

For now, the CanGraph software itself is not capable of analyzing the generated networks and providing answers to your questions. To remedy this as much as possible, this part of the tutorial is centered with giving meaningful examples of possible outputs of the program, making its results more understandable.

For this, we are using a sample CSV file (which you can get in the link below), the Results that you obtain when you compute that CSV file and the Neo4J browser. If you have, until now, only followed the steps of the tutorial regarding the installation and usage of the Apptainer image, you might not have neo4j installed in your system. The easiest way to solve this is to install it using *the python modules in CanGraph itself*. You can run the following set of commands on a Linux System:

```
sudo apt update # Update the software lisy to last version
# Install required software for python to run
sudo apt install python3.9 python3-pip default-jdk default-jre git
apt -y autoremove; apt -y clean # Clean the package manager
git clone https://github.com/OMB-IARC/CanGraph # Clone the project
cd CanGraph # CD into the directory
python3 -m pip --no-cache-dir install --upgrade pip # Upgrade PIP
python3 -m pip install -r requirements.txt # Install python modules
python3 setup.py --neo4j neo4j # And run it!
```

You can download the CSV file used for this examples here

And here, you can get the results for Metabolite 1 and Metabolite 2

2.4.1 Metabolite 1: *1-Amino-2-propanol*

For the first example, we have picked *1-Amino-2-propanol*, an amino-alcohol involved in the biosynthesis of cobalamin. This metabolite appeared during lab research at IARC, and was submitted to us to test the software and find more information on it. Originally, we were given 7 identifiers, from which our program can just pick 4 (Name,InChI,HMDB_ID & ChEBI_ID) to find synonyms using varios web services. Thanks to them, we were able to expand the search to 17 identifiers, which are synonyms of the starting 7.

Having obtained the results as explained in *the previous section*, we would now like to analyze the outputs using Neo4J browser (the program CanGraph is built for), which you have already installed in your software. To launch a new session, simply call:

```
./neo4j/bin/neo4j start
```

from the directory that contains the neo4j folder. A message will indicate the directories in use, and will show you the adress for Neo4J's web interface; most likely, http://localhost:7474. You should visit that link and log-in to the database (sometimes the login is automatic, in which case you have to do nothing) by using your user:password combination; the default ones are neo4j:neo4j, but, if you installed it using CanGraph scripts, you should try neo4j:$(head -n 1 .neo4jpassword); that is, neo4j as the user, and the first line of the `.neo4jpassword` the program will have created in your WorkDir as the password

Once logged-in, you will see a series of cards and a blinking cursor on the first; there, you can write your commands of choice in Neo4J's Cypher Query Language. What follows is a list of possible commands that might help you understand the usefulness and powerfulness of the program.

First, of course, we need to import the GRAPHML file; to do this, we place `metabolite_1.graphml` in `./neo4j/import`, and run:

```
CALL apoc.import.graphml("metabolite_1.graphml",
                         {batchSize: 5, useTypes:true, storeNodeIds:false,␣
↪readLabels:True,
                          useOptimizations: {type: "UNWIND_BATCH", unwindBatchSize: 5} })
```

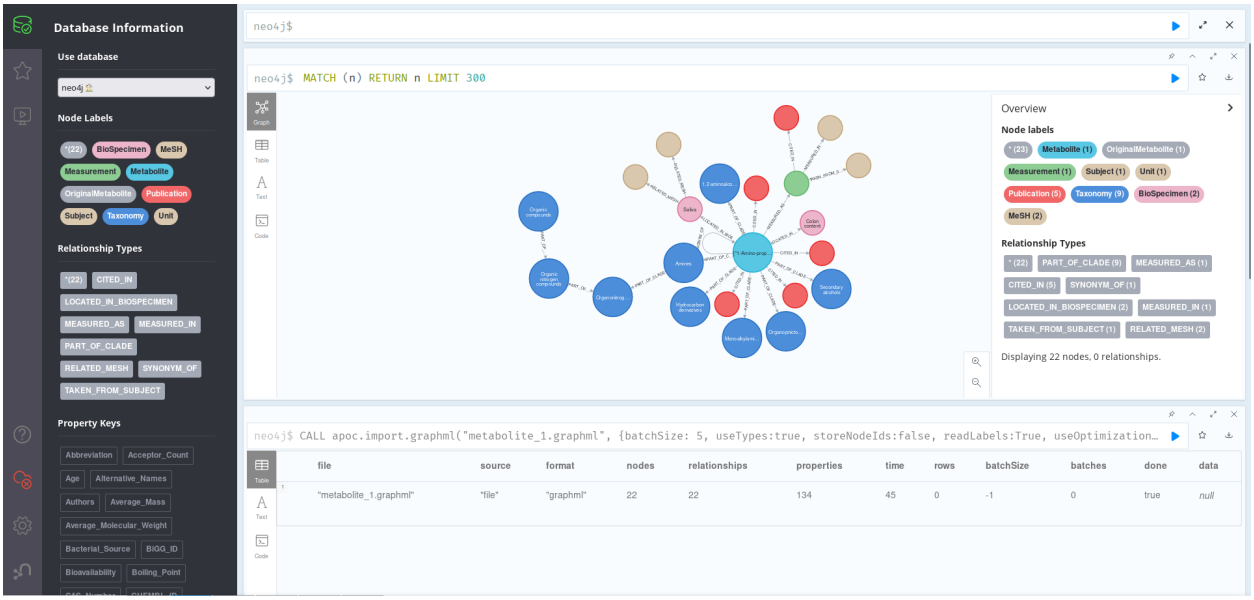
Inmediately, you will see the left tab populate with all the freshly imported data; in this case, 22 nodes, 22 relations and 134 properties, which represent all the knowledge available on *1-Amino-2-propanol* and any *really similar metabolites* on the 7 databases CanGraph is able to query. We can start exploring this graph by running:

```
MATCH (n) RETURN n LIMIT 300
```

This Cypher query asks Neo4J to display all nodes present in the database, regardless of their labels or connections. To make sure the program does not crash, we have attached a `LIMIT 300` statement to the query; in case your database is bigger than that (you can see the total number of nodes on the top left of the web app), you can also:

```
CALL db.schema.visualization()
```

which will just show you the *schema*, or general representation, of your database, so that you can see which kind of nodes have been imported. By sheer probability, the more nodes that your GraphML has, the more the schema will look similar to the complete schema as documented in the git repo



As we can see in the image above, the central *Metabolite* node above is related to 21 other nodes, which give us varied information on the *Metabolite* itself. For example, we can see that it can be found in *BioSpecimens Saliva* (with

MeSH_IDs *D012463* and *M0019372*) and *Colon content*, for which no MeSH_IDs where found. We can more dive more deeply on the database by, for example, asking for information on related Publications:

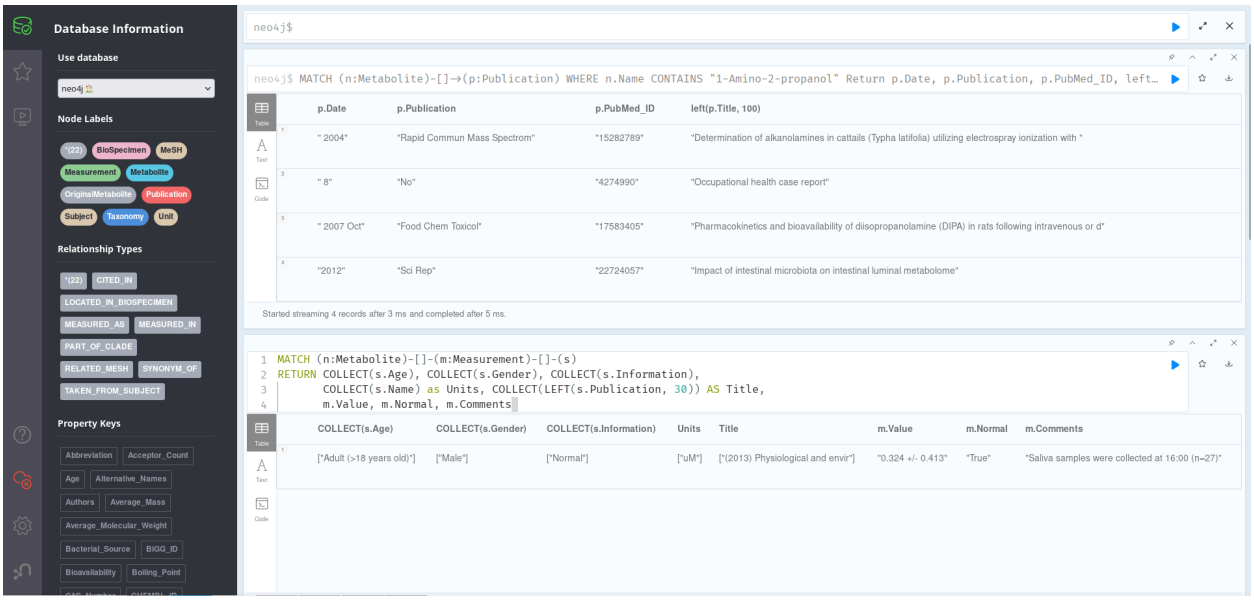
```
MATCH (n:Metabolite)-[]->(p:Publication)
WHERE n.Name CONTAINS "1-Amino-2-propanol"
Return p.Date, p.Publication, p.PubMed_ID, left(p.Title, 100)
```

Here, it is important to know that n.Name is a **list property**, which means that, instead of writing `WHERE n.Name = "1-Amino-2-propanol"`, we have to add a `CONTAINS` statement to the query. As we can see, there are 4 Publications related to our original *Metabolite*, all of them with a *Date*, *Publication*, *Title* (trimmed to 100 characters for visualization) and *PubMed_ID*.

We can also ask ourselves: where has this metabolite been found, exactly? One possible answer to this question is the query below:

```
MATCH (n:Metabolite)-[]-(m:Measurement)-[]-(s)
// Since there is only 1 metabolite, we dont need to filter by name
RETURN COLLECT(s.Age), COLLECT(s.Gender), COLLECT(s.Information),
       COLLECT(s.Name) as Units, COLLECT(LEFT(s.Publication, 30)) AS Title,
       m.Value, m.Normal, m.Comments
```

Here, we are getting all the "Measurement" nodes relating a Metabolite and some other nodes, and showing their properties; since we only have one of each, and in order to make the results look prettier, we are using `COLLECT` to turn properties into lists (and remove *null*). As we can see, this metabolite has been detected in male adults, obtaining a Normal value of 0.324 +/- 0.413 in a 2013 study. If we look more deeply at the Publication node where this information came from, we see that its properties are a bit messed up, which is why the year 2013 appears in the article's Title; this can happen sometimes when the material coming from the databases is not well standardised



Finally, we can also ask the database for info on the *kind of metabolite our metabolite is*, as in, which groups it could be categorized as. This can be found out thanks to the *Taxonomy* nodes. If we run the query below:

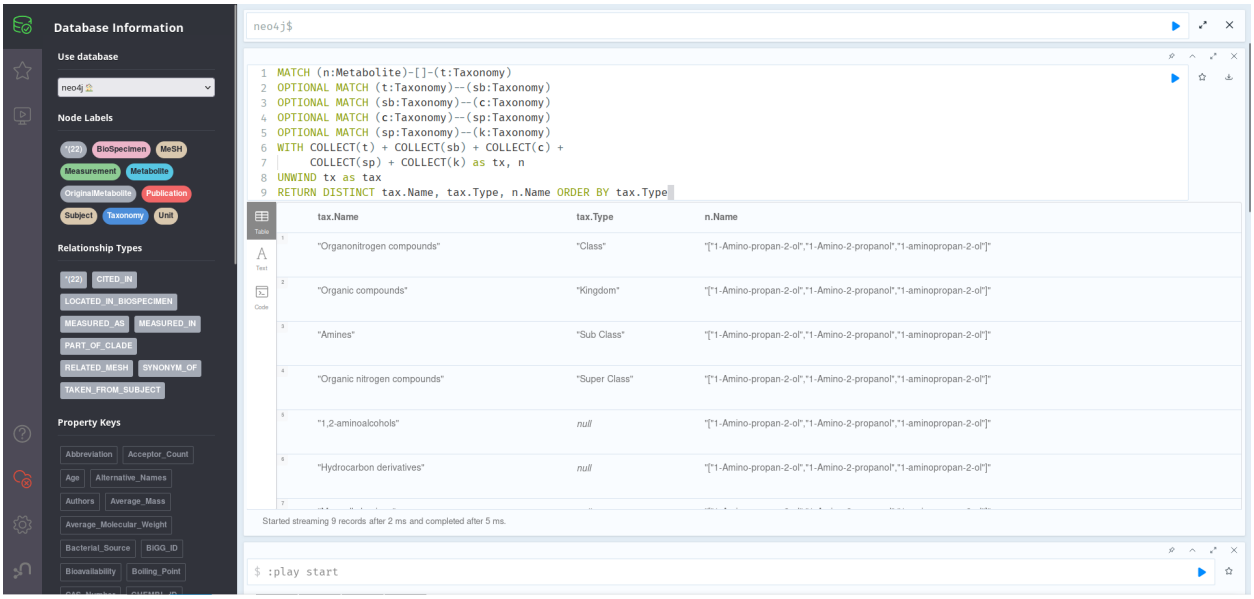
```
MATCH (n:Metabolite)-[]-(t:Taxonomy)
OPTIONAL MATCH (t:Taxonomy)--(sb:Taxonomy)
OPTIONAL MATCH (sb:Taxonomy)--(c:Taxonomy)
OPTIONAL MATCH (c:Taxonomy)--(sp:Taxonomy)
```

(continues on next page)

(continued from previous page)

```
OPTIONAL MATCH (sp:Taxonomy)--(k:Taxonomy)
WITH COLLECT(t) + COLLECT(sb) + COLLECT(c) +
     COLLECT(sp) + COLLECT(k) as tx, n
UNWIND tx as tax
RETURN DISTINCT tax.Name, tax.Type, n.Name
ORDER BY tax.Type
```

Here, we are getting all the taxonomies related to our main node, and those related to them up to 4 generations. Then, we are joining all the lists (collected using `COLLECT`), and unwinding them back in order to return each *Taxonomy*'s Name & Type, as well as the name of the metabolite they relate to (which stays constant since there is only 1). As we have seen before, here the *Name* property is a list

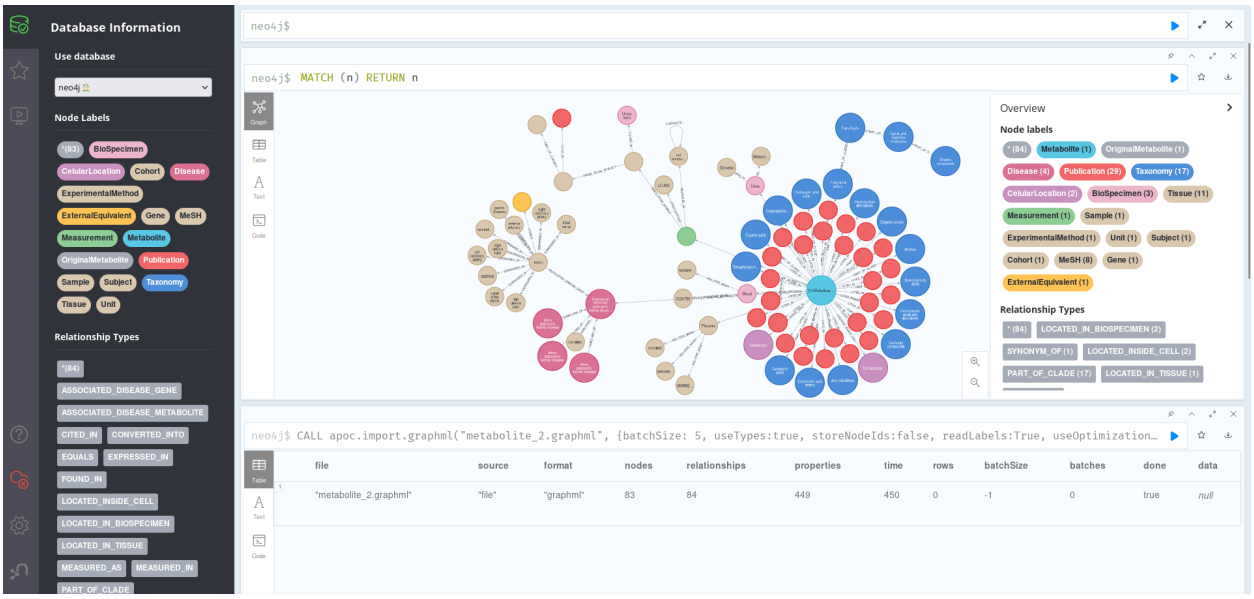


Once we are finished playing with the GraphML file, we can clean the database for the next example:

```
:auto MATCH (n)
        CALL { WITH n
        DETACH DELETE n
        } IN TRANSACTIONS OF 100 ROWS
```

2.4.2 Metabolite 2: *2-Methylbutyrylcarnitine*

For the second example, we picked *2-Methylbutyrylcarnitine*, a small molecule on which there is not much information online. The lab was able to provide us with 7 synonyms of the molecule, which our program expanded to 15 when doing the web search. After importing `metabolite_2.graphml` into a clean Neo4J session, as explained in the section above, we can proceed to explore it; since it consists of 83 nodes with 84 relations and 449 properties, printing all of them out is still a possibility:



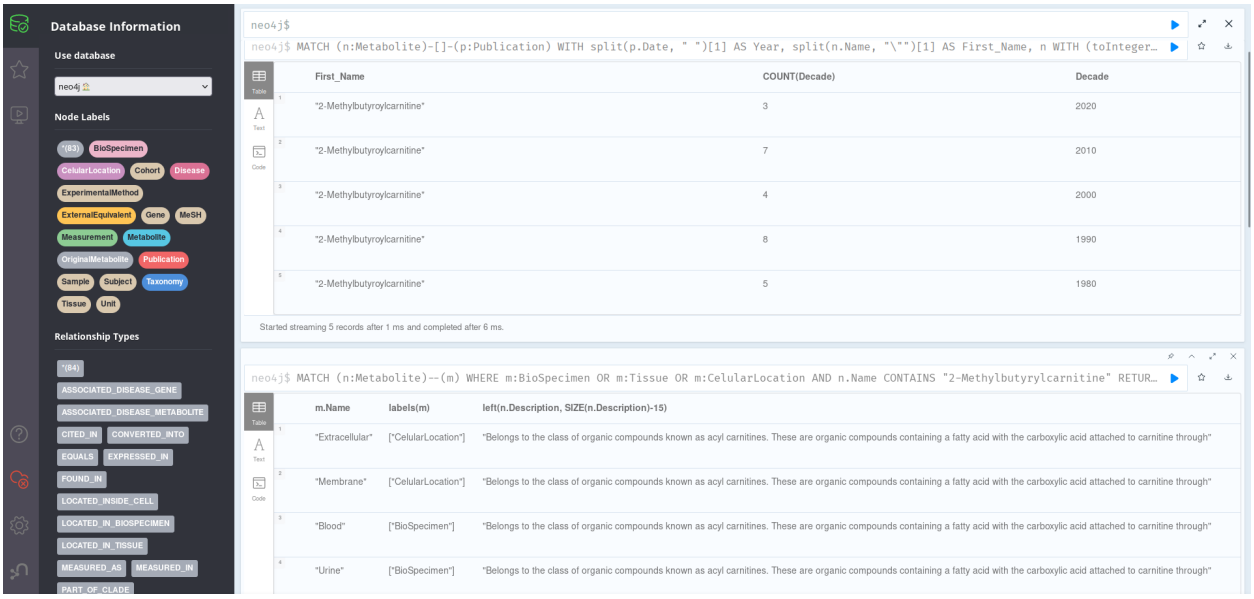
Here, we can once again see a central node (the *Metabolite*) with all the related information shown around it. Unlike in the other, however, there seem to be a lot of *Publication* nodes, which means some of them may be old and a bit outdated. We can find out by running:

```
MATCH (n:Metabolite)-[]-(p:Publication)
WITH split(p.Date, " ")[1] AS Year,
     split(n.Name, "\"")[1] AS First_Name, n
WITH (toInteger(Year)/10)*10 AS Decade, Year, First_Name, n
RETURN First_Name, COUNT(Decade), Decade
ORDER BY Decade DESC
```

This query groups *Publication*s related to *Metabolite*s by decade of publication, ordering them in reverse cronological order. We can see that, although the information is pretty much evenly split across decades, there are also recent articles (7 in the 2010s), which should mean this is an still-relevant metabolite. To find about its location in the organism (and thus, maybe, its function), we can run:

```
MATCH (n:Metabolite)-->(m)
WHERE m:BioSpecimen OR m:Tissue OR m:CelularLocation
AND n.Name CONTAINS "2-Methylbutyrylcarnitine"
RETURN m.Name, labels(m), left(n.Description, SIZE(n.Description)-15)
```

This shows that the metabolite is present mainly on the membrane and extracellular compartments of cells of the placenta, being detected in blood and urine. If we cheat a bit by adding a (truncated for the screenshot) `n.Description` column, we can see that this molecule is part of the *acyl carnitine* family, a group of organic compounds containing a fatty acid with the carboxylic acid attached to carnitine through an ester bond. We could further explore the groups this belongs to by running the query in the previous section.



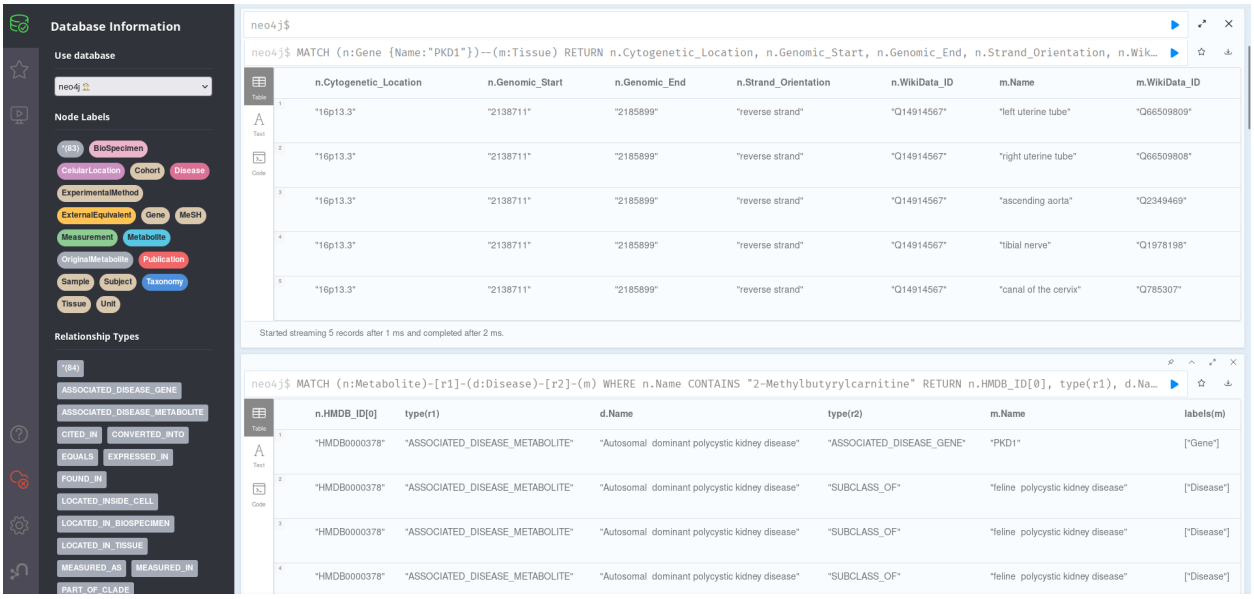
Now, we might want to find interesting information on possible disease associations; this can be done by running:

```
MATCH (n:Metabolite)-[r1]-(d:Disease)-[r2]-(m)
WHERE n.Name CONTAINS "2-Methylbutyrylcarnitine"
RETURN n.HMDB_ID[0], type(r1), d.Name, type(r2), m.Name, labels(m)
```

Here, we can see that the main *Metabolite* node is directly related to just one *Disease*: *Autosomal dominant polycystic kidney disease*. This, however, is a superclass of *eline polycystic kidney disease*, another disease; and is related with *MeSH_ID D016891*. We can also see a relation to gene *PKD1* about which we can learn a bit more by examining it on the first query of the section, the full grid of nodes. Since it seems to only be related to Tissue nodes, we can customize our queries to just return those ones:

```
MATCH (n:Gene {Name:"PKD1"})--(m:Tissue)
RETURN n.Cytogenetic_Location, n.Genomic_Start,
       n.Genomic_End, n.Strand_Orientation,
       n.WikiData_ID, m.Name, m.WikiData_ID
```

We can see that the *PKD1* gene is present in the 16th human chromosome, and that it has a length (end-start) of 47.188 base pairs on the reverse stand. Its transcripts are present in tissues such as *canal of the cervix*, *tibial nerve*, *ascending aorta* or the uterine tubes.



2.4.3 Topic Search: *Hepatocellular Carcinoma*

Finally, for a third and shorter example, we have taken a *topic search* on *Hepatocellular Carcinoma*. This is so because we wanted to illustrate that, although CanGraph has been devised **specifically** for the search of information on **specific metabolites**, it is possible to use it also for generic information searches on a topic of interest.

To do this, we have two options:

- We can either just provide the search term itself (HCC) together with some MeSH_IDs, using the **–noindex** flag. This is faster, but imports little nodes and is, in general, less useful: we know that the metabolites have "Hepatocellular Carcinoma" somewhere in their description or associated metadata, but we dont know exactly how tangencial the relation is or how much the import makes sense. Also, since the program only finds *exact text matches*, a disordered text chain (such as *Carcinoma, Hepatocellular* would not much
- Alternatively, we can do an *indirect search*, querying the program for a series of metabolites that we know to be related to HCC, *as well as* the text-chain "Hepatocellular Carcinoma" itself and its *Mesh_ID*, using the **indexed** version of the software. The metabolites we picked where the 18 present in the Publication called "Prediagnostic alterations in circulating bile acid profiles in the development of hepatocellular carcinoma", with doi:10.1002/ijc.33885

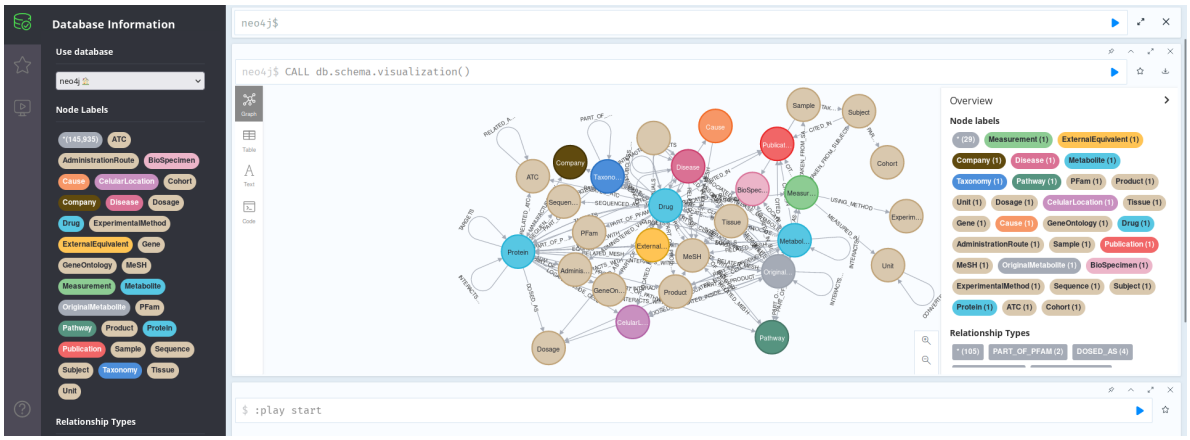
To make the results easier to follow, we are continuing this example with the **indexed** version. However, due to the confidentiality of some of the databases that we are using, and the fact that the exports for this example include thousands of nodes, we have decided not to provide the GraphML files for you to interact with; you can thus either generate them yourself, or just follow this tutorial for information.

From the 18 metabolites present in the cited paper, as well as the search term *Hepatocellular Carcinoma* and its MeSH_ID, *D015179*, CanGraph is able to extract 130 synonyms and related metabolites. After importing its results, following the steps described in the previous examples, we see that there are 145.935 nodes, 811.167 relationships and 1.319.878 properties, which will be the equivalent of the *Metabolite_4* export you would get when running our sample CSV.

Since this is a huge export, printing all the nodes in Neo4J Browser is not a good idea; trying to do it would crash neo4j itself. Thus, we have to come up with different solutions:

- First, we can print the schema, as previously explained:

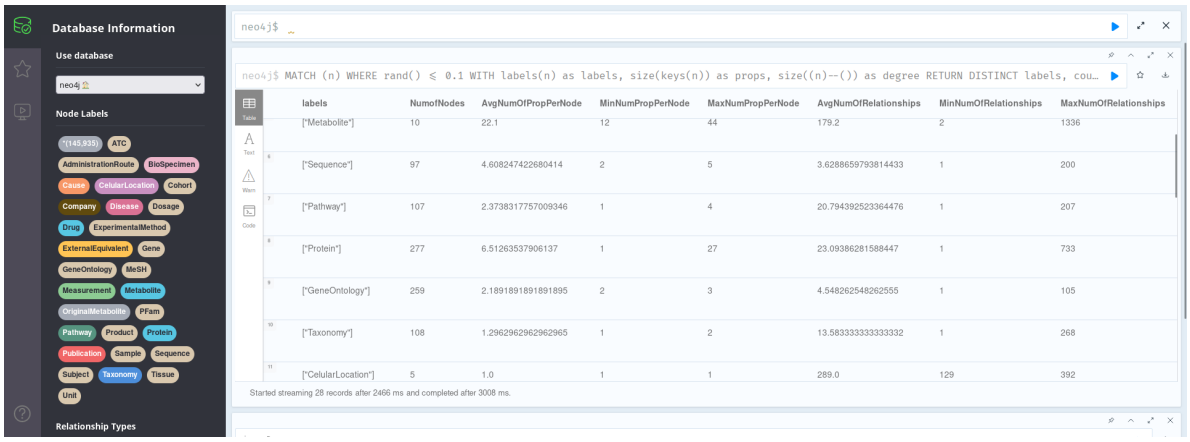
```
CALL db.schema.visualization()
```



As we can see in the image above, with so many nodes, this is not that useful. We can see that the Schema here conforms to the general schema as documented in the git repo, as it should, but the information is too squeezed to be useful: we need another approach.

- Using the following query we may print some interesting stats on the database itself:

```
MATCH (n)
WHERE rand() <= 0.1 // Select only 10% of nodes for speed
WITH labels(n) as labels, size(keys(n)) as props, size((n)--()) as degree
RETURN
DISTINCT labels, // Comment this line to show stats for all nodes
count(*) AS NumofNodes,
avg(props) AS AvgNumOfPropPerNode,
min(props) AS MinNumPropPerNode,
max(props) AS MaxNumPropPerNode,
avg(degree) AS AvgNumOfRelationships,
min(degree) AS MinNumOfRelationships,
max(degree) AS MaxNumOfRelationships
```

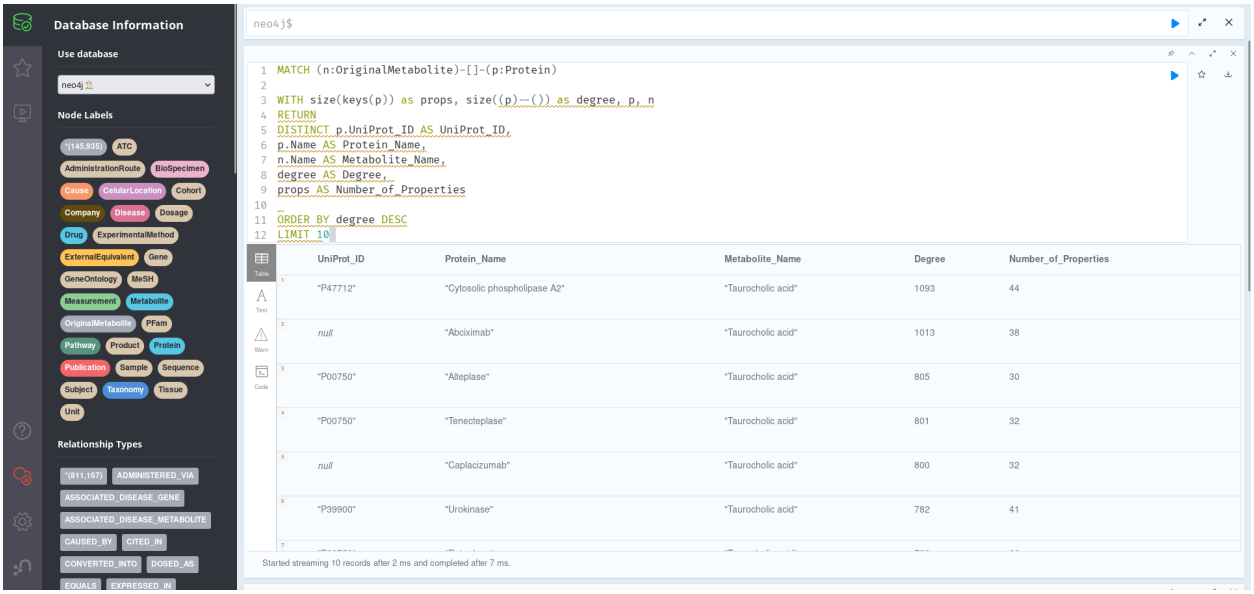


In the image above, we can see that, for instance, each metabolite has an average of 22 properties, with values ranging from 12-44 properties for each of the 10 nodes present in the database; or that there are 227 proteins, with an average of 23.09 proteins.

However, all this information still seems to generic, not providing any actionable insights into our data. We can

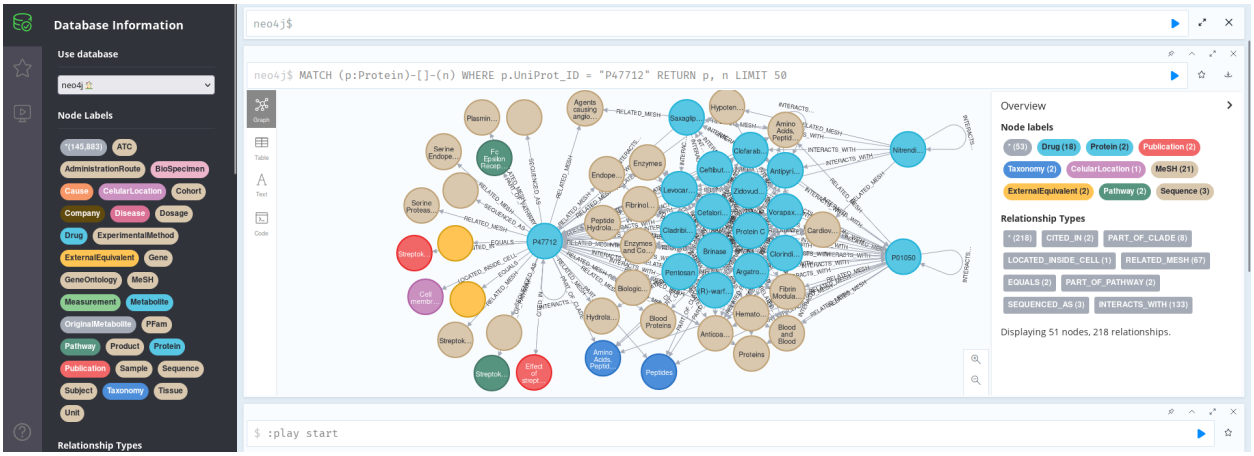
thus modify the previous query; for instance, to show us the 10 most important Proteins related to any of our `OriginalMetabolites` (the ones we took from the International Journal of Cancer paper). This *importance* characteristic can be calculated in a number of ways, but, as a proxy, here we are using the *degree* of the protein node; that is, the number of connections that it has to other nodes:

```
MATCH (n:OriginalMetabolite)-[]-(p:Protein)

WITH size(keys(p)) as props, size((p)--()) as degree, p, n
RETURN
DISTINCT p.UniProt_ID AS UniProt_ID,
p.Name AS Protein_Name,
n.Name AS Metabolite_Name,
degree AS Degree,
props AS Number_of_Properties

ORDER BY degree DESC
LIMIT 10
```



As we can see, most of the most important proteins are related to *Taurocholic acid*, which will probably be at least the most well-studied of our provided list of metabolites. Of these, the most relevant protein is *Cytosolic phospholipase A2*, with *UniProt_ID: P47712*. It may be worth exploring this protein further:

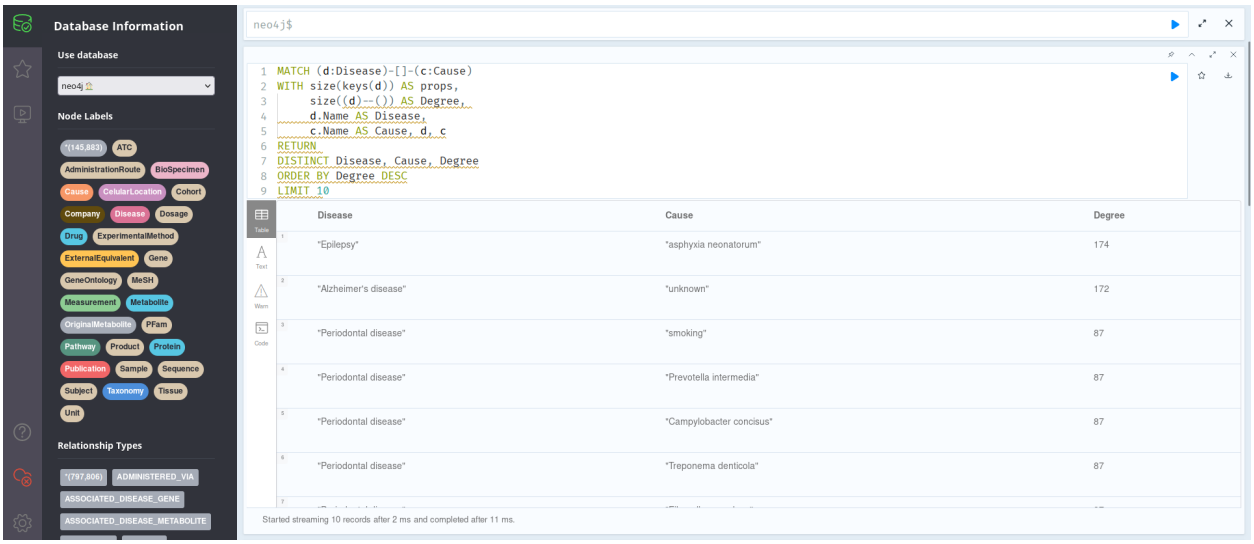
```
MATCH (p:Protein)-[]-(n)
WHERE p.UniProt_ID = "P47712"
RETURN p, n LIMIT 50
```



We can see that this protein is related with *Saxagliptin*, since they are both *Agents causing angioedema*; that is was cited in *U.S. Patent US6087332*, or that it appears on the *Cell Membrane* and is part of the *Streptokinase Action Pathway*.

Finally, we may display all the Diseases **which have known causes** that are present in our database (and which are somewhat related to HCC), ranked by degree:

```
MATCH (d:Disease)-[]-(c:Cause)
WITH size(keys(d)) AS props,
     size((d)--()) AS Degree,
     d.Name AS Disease,
     c.Name AS Cause, d, c
RETURN
DISTINCT Disease, Cause, Degree
ORDER BY Degree DESC
LIMIT 10
```



As one can see in the photo above, the most cited Diseases are *Epilepsy* and *Alzheimer's Disease*; this might be useful for your research, although one should proceed with caution; this does not *necessarily* mean that HCC is associated with Alzheimer's, but rather that they usually appear cited together in the 7 databases we are working with. Of course, this could also be an artifact, such as a given metabolite having many citations to, for instance, *Epilepsy*, and thus skewing the data when the other metabolites might not be linked with it; further exploring the Knowledge Graph, using the information provided on this tutorial and available on the internet, could help settle these and other questions.

2.5 Other Reading Material

There are a thousand ways that Knowledge Graphs, such as the ones *CanGraph* provides, can be explored. In this tutorial, we aim to provide a useful compilation of possible queries, ways of working, and forms of analyzing them, but this are in no way all the different ways meaningful insights can be extracted from each GraphML export *CanGraph* might provide. Neo4J is a vast software, and tons of other useful programs for graph manipulation exist, such as Cytoscape or python itself. Neo4J even provides its own data visualization tool, Neo4J Bloom, although it is not open source and might require a paid license.

We would thus like to recommend you the following, useful resources:

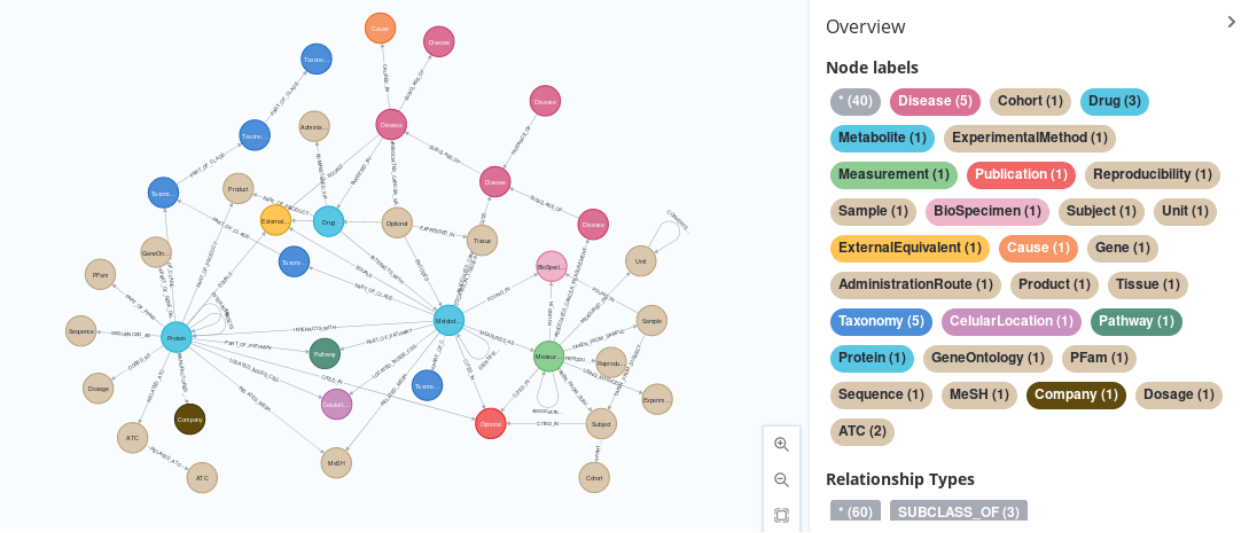
- Neo4J's Graph Academy hosts a free knowledge base on the field of Graphs, which can serve as useful tutorials in case what is reflected here is too confusing or difficult to follow. In particular, I would recommend:
  - Their Neo4j Fundamentals course for a primer on how graphs work and on Neo4J itself
  - Their Cypher Fundamentals course, to expand on this tutorial and on outputs processing
  - Their Building Neo4j Applications with Python course, in case you want to really dive into how CanGraph works and modify it according to your needs.
- Stack Overflow, which describes itself as "where developers learn, share & build careers" is a massive repository of questions and answers regarding all kinds of things in the informatics world. It is ikely that most of the questions you may have, or things you may want to use neo4j for, has been asked there already, so do not hesitate to search for more info or even ask there if you please.

Finally, in case you find any errors in this guides, or if you want to add any comments/suggestions, you may open an issue in our GitHub Repo. We look forward to any comments you may want to provide, and hope you have a fantastic experience using our software! :P

CHAPTER

THREE

CANGRAPH PACKAGE



This Git Project, created as part of my Master's Intenship at IARC, contains a series of scripts that pulls information from a series of **five** databases from their native format (XML, CSV, etc) into a common, GraphML format, using a shared schema that has been defined to minimize the number of repeated nodes and properties. This databases are:

- Exposome-Explorer: A hand-curated, high-quality database of associations between metabolites, food intakes and outakes and different diseases, specially cancers.
- Human Metabolome DataBase: An detailed, electronic database containing detailed information about small molecule metabolites found in the human body.
- DrugBank: A unique bioinformatics and cheminformatics resource that combines detailed drug data with comprehensive drug target information.
- Small Molecule Pathway Database: An interactive database containing more than 618 small molecule pathways found in humans, More than 70% of which are unique to this DB
- WikiData: The world's largest collaboratively generated collection of Open Data worldwide.

Each of then have their unique advantages and disadvantages (size, quality, etc) but they have been chosen to work together and help in identifying metabolites and their potential cancer associations at IARC.

With regards to the schema, it can be consulted in detail in the `new-schema.graphml` file, which can itself be opened in Neo4J by calling: `CALL apoc.import.graphml("new-schema.graphml", {useTypes:true, storeNodeIds:false, readLabels:True})` after placing it in your Neo4J's import directory (you can find it in the settings shown after starting the server with `sudo neo4j start`). It consists of a simplification of all the nodes present on the `old-schema.graphml` file (which itself represents the five different schemas that our five databases natively presented), arrived at by merging nodes and changing relationship names so that they are unique (and, thus,

more actionable). One property, `LabelName` has been added as a dummy name to generate the image you can see in the header.

This repo contains two kind of scripts: first, some `build_database.py` scripts, which contain the information to re-build the databases in the common format from scratch, and are located in subsequent subfolders named after the database they come from (more info can be consulted on them on their respective READMEs) and a common `main.py` script, which can be used to query for sub-networks based solely on info presented on a **sample_input.csv** database of identified compounds which we would like to annotate.

3.1 Intallation

To use this script, you should first clone it into your personal computer. The easiest way to do this is to git clone the repo:

1. Install git (if not already installed) and other requirements. On linux: `sudo apt install git curl`
2. Clone the repo: `git clone https://codeberg.org/FlyingFlamingo/graphify-databases`
3. Step into the directory `cd graphify-databases`

Once the project has been installed, you **must** run `setup.py`, a preparation script that guides you through the process of installing all five databases on your computer, so that then we can correctly process them and generate the sub-networks. You should also install the required python modules and run the setup script:

4. PIP install all dependencies: `pip install -r requirements.txt`
5. Run the setup script: `python3 setup.py`

Once this has been done, you are ready to start using the main script!

NOTE: If you do not wish to use git, you can manually download the repo by clicking here

3.2 Usage

To generate this sub-networks (the original idea of the project) you should run:

```
python3 main.py neo4jadress databaseusername databasepassword databasefolder inputfile
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **username**: the username for your neo4j instance. Remember, the default is neo4j
- **password**: the password for your database. Since the arguments are passed by BaSH onto python3, you might need to escape special characters
- **databasefolder**: The folder indicated to `setup.py` as the one where your databases will be stored
- **inputfile**: The location of the CSV file in which the program will search for metabolites. This file should be a Comma-Separated file, with the following format: `MonoisotopicMass, SMILES, InChIKey, Name, InChI, Identifier, ChEBI`

NOTE: When committing to the repo, try to use GitMojis to illustrate your commit :p

3.3 Important Notices

- Some databases are auto-integrated based on their URLs. This URLs, as well as those of existing dependencies, may change over time. Please make sure to have them updated in case you want to run the latest version of the databases
- We have made our best efforts to make the script as multi-platform as possible; however, the script has been developed with Linux in mind, and you may need to install additional packages if you want to run it on Windows or MacOS. Please, check the `dependencies` section for more info

3.4 Dependencies

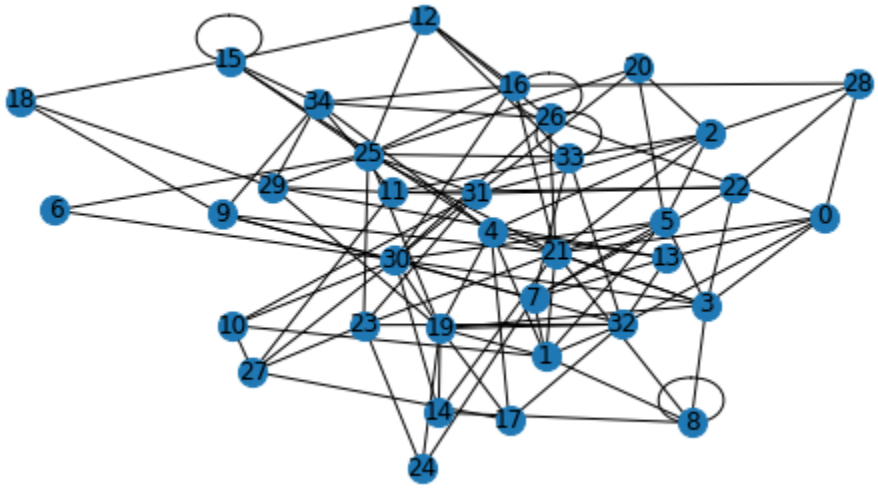
This python package has the following known dependencies:

Package	Description	Order to install
Python 3.8	the python programming language	`sudo apt install python3`
cURL	command line tool for transferring data from URLs	`sudo apt install curl`
neo4j	a graph dbms	`python3 -c'import setup; setup.setup_neo4j("neo4j", True)'`

Alternatively, as a one-liner: `sudo apt install python3 curl; python3 -c'import setup; setup.setup_neo4j("neo4j", True)`

3.5 Subpackages

3.5.1 CanGraph.ExposomeExplorer package



This package, created as part of my Master's Intenship at IARC, transitions the exposome-explorer database (a high quality, hand-curated database containing associations of foods and chemical compounds with cancer) to Neo4J format in an automated way, providing an export in GraphML format.

To run, it uses `alive_progress` to generate an interactive progress bar (that shows the script is still running through its most time-consuming parts) and the `neo4j` python driver. This requirements can be installed using: `pip install -r requirements.txt`.

To run the script itself, use:

```
python3 main.py neo4jadress databasename databasepassword csvfolder
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **databasename**: the name of the database in use. If using the free version, there will only be one database per project (neo4j being the default name); if using the pro version, you can specify an alternate name here
- **databasepassword**: the passowrd for the **databasename** DataBase. Since the arguments are passed by BaSH onto python3, you might need to escape special characters
- **csvfolder**: The folder where the CSV files for the Exposome Explorer database are stored. This CSVs have to be manually exported from the (confidential) database itself, and are **NOT** equivalent to those found in exposome-explorer download's page

An archived version of this repository that takes into account the gitignored files can be created using: `git archive HEAD -o ${PWD##*/}.zip`

The package consists of the following modules:

CanGraph.ExposomeExplorer.build_database module

A python module that provides the necessary functions to transition the Exposome Explorer database to graph format, either from scratch importing all the nodes (as showcased in *CanGraph.ExposomeExplorer.main*) or in a case-by-case basis, to annotate existing metabolites (as showcased in *CanGraph.main*).

`add_cancer_associations`(*filename*)

Imports the 'cancer_associations' database as a relation between a given Cancer and a Measurement

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_components`(*filename*)

Adds "Metabolite" nodes from Exposome-Explorer's components.csv This is because this components are, in fact, metabolites, either from food or from human metabolism

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_correlations`(*filename*)

Imports the 'correlations' database as a relation between two measurements: the intake_id, a food taken by the organism and registered using dietary questionnaires and the excretion_id, a chemical found in human biological samples, such that, when one takes one component, one will excrete the other. Data comes from epidemiological studies where dietary questionnaires are administered, and biomarkers are measured in specimens

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_measurements_stuff`(*filename*)

A massive and slow-running function that creates ALL the relations between the 'measurements' table and all other related tables:

- units: The units in which a given measurement is expressed
- components: The component which is being measured
- samples: The sample from which a measurement is taken
- experimental_methods: The method used to take a measurement

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_metabolomic_associations`(*filename*)

Imports the 'metabolomic_associations' database as a relation between to measurements: the intake_id, a food taken by the organism and registered using dietary questionnaires and the excretion_id, a chemical found in human biological samples, such that, when one takes one component, one will excrete the other. Data comes from Metabolomics studies seeking to identify putative dietary biomarkers.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_microbial_metabolite_identifications`(*filename*)

Imports the relations pertaining to the "microbial_metabolite_identifications" table. A component (i.e. a metabolite) can be identified as a Microbial Metabolite, which means it has an equivalent in the microbiome. This can have a given reference and a tissue (BioSpecimen) in which it occurs.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running

- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_reproducibilities`(*filename*)

Creates relations between the "reproducibilities" and the "measurements" table, using "initial_id", an old identifier, for the linkage

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_samples`(*filename*)

Imports the relations pertaining to the "samples" table. A sample will be taken from a given subject and a given tissue (that is, a specimen, which will be blood, urine, etc)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`add_subjects`(*filename*)

Imports the relations pertaining to the "subjects" table. Basically, a subject can appear in a given publication, and will be part of a cohort (i.e. a grop of subjects)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`annotate_auto_units`(*filename*)

Shows the correlations between two units, converted using the rubygem 'https://github.com/masa16/phys-units' which standarizes units of measurement for our data

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

`annotate_cancers`(*filename*)

Adds "Cancer" nodes from Exposome-Explorer's cancers.csv

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_cohorts(*filename*)

Adds "Cohort" nodes from Exposome-Explorer's cohorts.csv

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_experimental_methods(*filename*)

Adds "ExperimentalMethod" nodes from Exposome-Explorer's experimental_methods.csv

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_measurements(*filename*)

Adds "Measurement" nodes from Exposome-Explorer's measurements.csv

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_microbial_metabolite_info(*filename*)

Adds "Metabolite" nodes from Exposome-Explorer's microbial_metabolite_identifications.csv These represent all metabolites that have been re-identified as present, for instance, in the microbiome.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_publications(*filename*)

Adds "Publication" nodes from Exposome-Explorer's publications.csv

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running

- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_reproducibilities(*filename*)

Adds "Reproducibility" nodes from Exposome-Explorer's reproducibilities.csv These represent the conditions under which a given study/measurement was carried

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_samples(*filename*)

Adds "Sample" nodes from Exposome-Explorer's samples.csv From a Sample, one can take a series of measurements

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_specimens(*filename*)

Annotates "BioSpecimen" nodes from Exposome-Explorer's specimens.csv whose ID is already present on the DB A biospecimen is a type of tissue where a measurement can originate, such as orine, csf fluid, etc

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_subjects(*filename*)

Annotates "Subject" nodes from Exposome-Explorer's subjects.csv whose ID is already present on the DB

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

annotate_units(*filename*)

Adds "Unit" nodes from Exposome-Explorer's units.csv A unit can be converted into other (for example, for normalization)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

build_from_file(*databasepath*, *Neo4JImportPath*, *driver*, *bar=None*, *do_all=False*, *keep_counts_and_displayeds=True*, *keep_cross_properties=False*)

A function able to build a portion of the Exposome-Explorer database in graph format, provided that at least one "Component" (Metabolite) node is present in said database. It works by using that node as an starting point from which to search in the rest of the Exposome_Explorer database, finding related nodes there.

Parameters

- **databasepath** (*str*) – The path to the database where all Exposome-Explorer CSVs are stored
- **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **bar** – The bar() object from alive_bar, in case we want the function to run with do_all=True
- **do_all** (*bool*) – True if importing the whole database; False if just importing a part of it
- **keep_counts_and_displayeds** (*bool*) – Whether to keep the properties ending with `_count` & `displayed_` that, although present in the original DB, might be considered not useful for us.
- **keep_cross_properties** (*bool*) – Whether to keep the properties used to cross-reference in the original Neo4J database.

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

Note: This wont work if a "Component" (Metabolite) node is not already present; when building the database, either full or by parts, you should import the respective Components first

---

Warning: Due to the script's design, only nodes which have a connection to nodes previously present on the database will be imported. This is on purpose: unconnected nodes don't mean much in a Graph DataBase

import_csv(*filename*, *label*)

Imports a given CSV into Neo4J. This CSV **must** be present in Neo4J's Import Path

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported
- **label** (*str*) – The label of the Neo4J nodes that will be imported, with the columns of the CSV being its properties.

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

Note: For this to work, you HAVE TO have APOC availaible on your Neo4J installation

`remove_counts_and_displayeds`(*inputfile*, *outputfile*)

Removes `_count` & `displayed_` text-strings from a given file, so that, when processing it with the other functions present in this document, they ignore the columns containing said text-strings, which represent properties which are considered not useful for our program. This is. of course, not the most elegant, but it works.

Parameters

- **inputfile** (*str*) – The path to the file from which `_count` & `displayed_ text-strings are to be removed
- **outputfile** (*str*) – The path of the file where the contents of the replaced file will be written.

Returns The function does not have a return; instead, it transforms ``inputfile`` into `outputfile`

`remove_cross_properties`()

Removes some properties that were added by the other functions present in this script, that are used to cross-reference the different tables in the Relational Database EE comes from, and that, in a Graph Database, are no longer necessary.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it accordingly.

Return type neo4j.Result

CanGraph.ExposomeExplorer.main module

A python module that leverages the functions present in the *build_database* module to recreate the exposome-explorer database using a graph format and Neo4J, and then provides an GraphML export file.

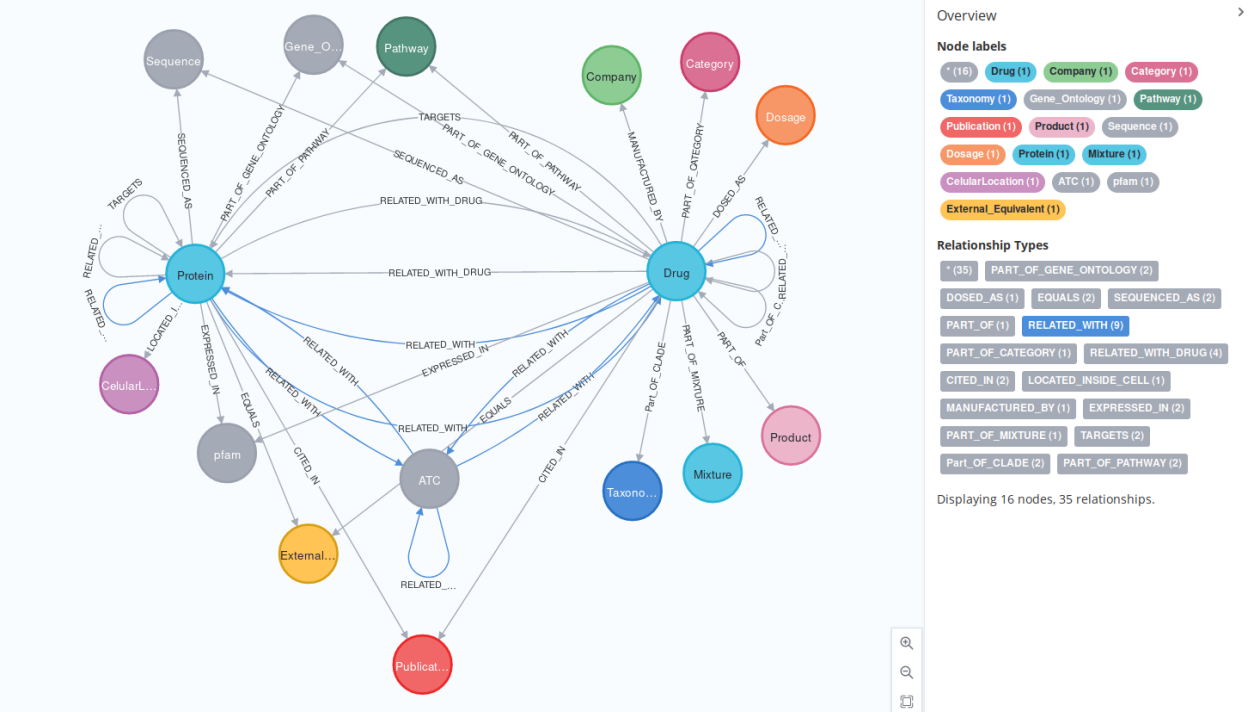
Please note that, to work, the functions here pre-suppose you have access to Exposome-Explorer internal CSVs, and that you have placed them under a folder provided as `sys.argv[4]`. These CSVs are confidential, and can only be accessed under request to the *International Agency for Research on Cancer*.

For more details on how to run this script, please consult the package's README

`main`(*args*)

The function that executes the code

3.5.2 CanGraph.GraphifyDrugBank package



This package, created as part of my Master's Intenship at IARC, imports nodes from the DrugBank Database (a high quality database containing a drugs and proteins with some characteristics) to Neo4J format in an automated way, providing an export in GraphML format.

To run, it uses `alive_progress` to generate an interactive progress bar (that shows the script is still running through its most time-consuming parts) and the `neo4j` python driver. This requirements can be installed using: `pip install -r requirements.txt`.

To run the script itself, use:

```
python3 main.py neo4jadress username databasepassword filename
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **username**: the username for your neo4j instance. Remember, the default is neo4j
- **password**: the passowrd for your database. Since the arguments are passed by BaSH onto python3, you might need to escape special characters
- **filename**: the database's file name. For practical purpouses, it **must** be present in ./xmlfolder

Please note that there are two kinds of functions in the associated code: those that use python f-strings, which themselves contain text that *cannot* be directly copied into Neo4J (for instance, double brackets have to be turned into simple brackets) and normal multi-line strings, which can. This is because f-strings allow for variable customization, while normal strings dont.

An archived version of this repository that takes into account the gitignored files can be created using: `git archive HEAD -o ${PWD##*/}.zip`

Important Notices

- To access the Drugbank database, you have to previously request access at: https://go.drugbank.com/public_users/sign_up
- Some XML tags have been intentionally not processed; for example, the tag didn't seem to dd anything new, and the and tags are likely not relevant for our project. The tag has no info (check `cat full\ database.xml | grep "<ahfs-codes>"`) and the tag seemed like too much work (same for). nOT USE because not of use to us
- Some :Proteins might also be :Drugs and viceversa: this makes the schema difficult to understand, but provides more meaningful data. This is also why some relations (`RELATED_WITH`, for instance) dont have a relation: this way, we avoid duplicates.

The package consists of the following modules:

CanGraph.GraphifyDrugBank.build_database module

A python module that provides the necessary functions to transition the DrugBank database to graph format, either from scratch importing all the nodes (as showcased in *CanGraph.GraphifyDrugBank.main*) or in a case-by-case basis, to annotate existing metabolites (as showcased in *CanGraph.main*).

add_atc_codes(*filename*)

Creates "ATC" nodes based on XML files obtained from the DrugBank website. These represent the different ATC codes a Drug can be related with (including an small taxonomy)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

add_categories(*filename*)

Creates "Category" nodes based on XML files obtained from the DrugBank website. These represent the different MeSH IDs a Drug can be related with

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Each category seems to have an associated MeSH ID. Maybe could rename nodes as MeSH?

---

add_dosages(*filename*)

Creates "Dosage" nodes based on XML files obtained from the DrugBank website. These represent the different Dosages that a Drug should be administered at.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Warning: Using CREATE might generate duplicate nodes, but there was no unique characteristic to MERGE nodes into.

`add_drug_interactions`(*filename*)

Creates `(d)-[r:RELATED_WITH_DRUG]-(dd)` interactions between "Drug" nodes, whether they existed before or not. These are intentionally non-directional, as they should be related with each other.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

`add_drugs`(*filename*)

Creates "Drug" nodes based on XML files obtained from the DrugBank website, adding some essential identifiers and external properties.

See also:

This way of working has been taken from William Lyon's Blog

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Since Publications dont have any standard identificator, they are created using the "Title"

`add_experimental_properties`(*filename*)

Adds some experimental properties to existing "Drug" nodes based on XML files obtained from the DrugBank website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

`add_external_equivalents`(*filename*)

Adds some external equivalents to existing "Drug" nodes based on XML files obtained from the DrugBank website. This should be "exact matches" of the Drug in other databases.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: The main reason to add them as "External-Equivalents" is because I felt these IDs where of not much use (and are thus easier to eliminate due to their common label)

`add_external_identifiers`(*filename*)

Adds some external identifiers to existing "Drug" nodes based on XML files obtained from the DrugBank website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: These also adds a "Protein" label to any "Drug"-labeled nodes which have a "UniProtKB"-ID among their properties. NOTE that this can look confusing in the DB Schema!!!

`add_general_references`(*filename*)

Creates "Publication" nodes based on XML files obtained from the DrugBank website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Since not all nodes present a "PubMed_ID" field (which would be ideal to uniquely-identify Publications, as the "Text" field is way more prone to typos/errors), nodes will be created using the "Authors" field. This means some duplicates might exist, which should be accounted for.

---

add_manufacturers(*filename*)

Creates "Company" nodes based on XML files obtained from the DrugBank website. These represent the different Companies that manufacture a Drug's compound (not just package it)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

add_mixtures(*filename*)

Creates "Mixture" nodes based on XML files obtained from the DrugBank website. These are the mixtures of existing Drugs, which may or may not be on the market.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: This doesn't seem of much use, but has been added nonetheless just in case.

---

add_packagers(*filename*)

Creates "Company" nodes based on XML files obtained from the DrugBank website. These represent the different Companies that package a Drug's compounds (not the ones that manufacture them)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

add_pathways_and_relations(*filename*)

Adds "Pathway" nodes based on XML files obtained from the DrugBank website. It also adds some relations between Drugs and Proteins (which, remember, could even be the same kind of node) It is also able to tag both a Protein's and a Drug¡s relation with a given Pathway In general, a Pathway involves a collection of Enzymes, Drugs and Proteins, with a SMPDB_ID (cool for interconnexion!)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Warning: This function uses a "double UNWIND" clause, which means that we are only representing <pathways> tags with <enzymes> tags inside. Fortunately, this seems to seldom not happen, so it should represent no problem.

add_products(*filename*)

Creates "Product" nodes based on XML files obtained from the DrugBank website. These are the individual medicaments that have been approved (or not) by the FDA

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Warning: Using CREATE means that duplicates will appear; unfortunately, I couldnt any unique_id field to use as ID when MERGEing the nodes. This should be accounted for.

add_sequences(*filename*)

Creates "Sequence" nodes based on XML files obtained from the DrugBank website. These represent the AminoAcid sequence of Drugs that are of a peptidic nature.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Todo: In some other parts of the script, sequences are being added as properties on Protein nodes. A common format should be set.

add_targets_enzymes_carriers_and_transporters(*filename*, *tag_name*)

A *REALLY HUGE* function. It takes a filename and a tag_name, and gets info and creates "Protein" nodes with tag_name set as their role. It also adds a bunch of additional info, such as Publications, Targets, Actions, GO_IDs, PFAMs and/or some External IDs

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported
- **tag_name** (*str*) – The type of Protein node you want to import; it must be one of ["enzymes", "carriers", "transporters"] It is recommended that you run this function thrice, once for each type of protein

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Warning: We are using a bunch of concatenated UNWINDs, which force the existance of all elements in the UNWIND chain. This might remove some elements, but this is a HUUUUUGE database, and, to be honest, most things seem to almost always be present. An example is References and Polypeptides; Since there seem to be more References that Polypeptides, we try to UNWIND those first. The same can be said on the rest of UNWINDS: as we have external-id >>>>>>>>>> go-classifier >>>> pfam >> synonyms (in order of *occurrence*. not *number* of tags) , we UNWIND in that order to mitigate data loss

Note: To fix repetitions in properties such as Actions or Synonyms (caused by the HUGE number of UNWINDs), we tried lots of different strategies, finally coming up with SET p.Synonyms = replace(p.Synonyms, synonym._text + ",", ""). This is cool! But means there will always be a trailing comma (removing it was not easy in this same transaction, though it could (TODO?) be done at the end.

Note: <tag_name>

Todo: Investigate https://stackoverflow.com/questions/14026217/using-neo4j-distinct-and-order-by-on-different-properties

`add_taxonomy`(*filename*)

Creates "Taxonomy" nodes based on XML files obtained from the DrugBank website. These represent the "kind" of Drug we are dealing with (Family, etc)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: It only creates relationships in the Kingdom -> Super Class -> Class -> Subclass direction, and from any node -> Drug. This means that, if any member of the Kingdom -> Super Class -> Class -> Subclass is absent, the line will be broken; hopefully in that case a new Drug will come in to rescue and settle the relation!

Warning: Some nodes without labels might be created if names are null: This has to be accounted for later on in the process

`build_from_file`(*newfile*, *driver*)

A function able to build a portion of the DrugBank database in graph format, provided that one XML is supplied to it. This can either be the `full_database.xml` file that you can get in DrugBank's website, or a splitted version of it, with just one item per file (which is recommended due to memory limitations)

Parameters

- **newfile** (*str*) – The path of the XML file to import

- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

CanGraph.GraphifyDrugBank.main module

A python module that leverages the functions present in the *build_database* module to recreate the DrugBank database using a graph format and Neo4J, and then provides an GraphML export file.

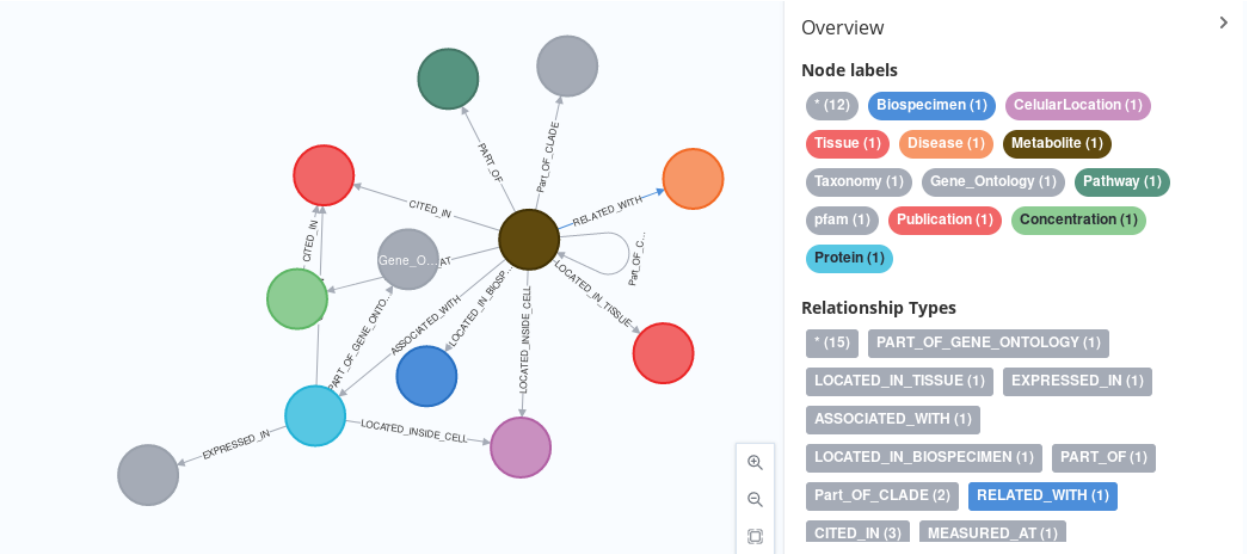
Please note that, to work, the functions here pre-suppose you have a DrugBank account with access to the whole Database. You have to prevously apply for it at: https://go.drugbank.com/public_users/sign_up, and place the `full_database.xml` file at `./xmlfolder`.

For more details on how to run this script, please consult the package's README

main()

The function that executes the code

3.5.3 CanGraph.GraphifyHMDB package



This script, created as part of my Master's Intenship at IARC, imports nodes from the Human Metabolome Database (a high quality, database containing a list of metabolites and proteins associated to different diseases) to Neo4J format in an automated way, providing an export in GraphML format.

To run, it uses `alive_progress` to generate an interactive progress bar (that shows the script is still running through its most time-consuming parts) and the `neo4j` python driver. This requirements can be installed using: `pip install -r requirements.txt`.

To run the script itself, use:

```
python3 main.py neo4jadress username databasepassword
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **username**: the username for your neo4j instance. Remember, the default is neo4j

- **password**: the passowrd for your database. Since the arguments are passed by BaSH onto python3, you might need to escape special characters

Please note that there are two kinds of functions in the associated code: those that use python f-strings, which themselves contain text that *cannot* be directly copied into Neo4J (for instance, double brackets have to be turned into simple brackets) and normal multi-line strings, which can. This is because f-strings allow for variable customization, while normal strings dont.

An archived version of this repository that takes into account the gitignored files can be created using: `git archive HEAD -o ${PWD##*/}.zip`

Important Notices

- Please ensure you have internet access, enough espace in your hard drive (around 5 GB) and read-write access in `./xmlfolder`. The files needed to build the database will be stored there.
- There are two kinds of high-level nodes stored in this database: "Metabolites", which are individual compounds present in the Human Metabolome; and "Proteins", which are normally enzimes and are related to one or multiple metabolites. There are different types of metabolites, but they were all imported in the same way; their origin can be differenced by the "" field on the corresponding "Concentration" nodes. You could run a query such as: `MATCH (n:Metabolite)-[r:MEASURED_AT]-(c:Concentration) RETURN DISTINCT c.Biospecimen`
- Some XML tags have been intentionally not processed; for example, the tag seemed like too much info unrelated to our project, or the tags, which could be useful but seemed to only link to external DBs

---

The package consists of the following modules:

CanGraph.GraphifyHMDB.build_database module

A python module that provides the necessary functions to transition the HMDB database to graph format, either from scratch importing all the nodes (as showcased in *CanGraph.GraphifyHMDB.main*) or in a case-by-case basis, to annotate existing metabolites (as showcased in *CanGraph.main*).

add_biological_properties(*filename*)

Adds biological properties to existing "Metabolite" nodes based on XML files obtained from the HMDB website. In this case, only properties labeled as <predicted_properties> are added.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Another option would have been to auto-add all the properties, and name them using RETURN "Predicted " + apoc.text.capitalizeAll(replace(kind, "_", " ")), value; however, this way we can select and not duplicate / overwrite values.

Todo: It would be nice to be able to distinguish between experimental and predicted properties

---

add_concentrations_abnormal(*filename*)

Creates "Concentration" nodes based on XML files obtained from the HMDB website. In this function, only metabolites that are labeled as "abnormal_concentration" are added.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Here, an UNWIND clause is used instead of a FOREACH clause. This provides better performance, since, unlike FOREACH, UNWIND does not process rows with empty values

---

Warning: Using the CREATE row forces the creation of a Concentration node, even when some values might be missing. However, this means some bogus nodes could be added, which MUST be accounted for at the end of the DB-Creation process.

add_concentrations_normal(*filename*)

Creates "Concentration" nodes based on XML files obtained from the HMDB website. In this function, only metabolites that are labeled as "normal_concentration" are added.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Here, an UNWIND clause is used instead of a FOREACH clause. This provides better performance, since, unlike FOREACH, UNWIND does not process rows with empty values

---

Warning: Using the CREATE row forces the creation of a Concentration node, even when some values might be missing. However, this means some bogus nodes could be added, which MUST be accounted for at the end of the DB-Creation process.

add_diseases(*filename*)

Creates "Publication" nodes based on XML files obtained from the HMDB website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Here, an UNWIND clause is used instead of a FOREACH clause. This provides better performance, since, unlike FOREACH, UNWIND does not process rows with empty values (and, logically, there should be no Publication if there is no Disease)

Note: Publications are created with a (m)-[r:CITED_IN]->(p) relation with Metabolite nodes. If one wants to find the Publication nodes related to a given Metabolite/Disease relation, one can use:

```
MATCH p=()-[r:RELATED_WITH]->()
  WITH split(r.PubMed_ID, ",") as pubmed
    UNWIND pubmed as find_this
    MATCH (p:Publication)
      WHERE p.PubMed_ID = find_this
RETURN p
```

`add_experimental_properties`(*filename*)

Adds properties to existing "Metabolite" nodes based on XML files obtained from the HMDB website. In this case, only properties labeled as <experimental_properties> are added.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Another option would have been to auto-add all the properties, and name them using RETURN "Experimental " + apoc.text.capitalizeAll(replace(kind, "_", " ")), value; however, this way we can select and not duplicate / overwrite values.

Todo: It would be nice to be able to distinguish between experimental and predicted properties

---

`add_gene_properties`(*filename*)

Adds some properties to existing "Protein" nodes based on XML files obtained from the HMDB website. In this case, properties will mostly relate to the gene from which the protein originates.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: We are not creating "Gene" nodes (even though each protein comes from a given gene) because we believe not enough information is being given about them.

`add_general_references`(*filename*, *type_of*)

Creates "Publication" nodes based on XML files obtained from the HMDB website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Since not all nodes present a "PubMed_ID" field (which would be ideal to uniquely-identify Publications, as the "Text" field is way more prone to typos/errors), nodes will be created using the "Authors" field. This means some duplicates might exist, which should be accounted for.

Note: Unlike the rest, here we are not matching metabolites, but ALSO proteins. This is intentional.

`add_go_classifications`(*filename*)

Creates "Gene Ontology" nodes based on XML files obtained from the HMDB website. This relates each protein to some GO-Terms

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

`add_metabolite_associations`(*filename*)

Adds associations contained in the "protein" file, between proteins and metabolites.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Like he "add_metabolite_associations" function, this creates non-directional relationships (m)-[r:ASSOCIATED_WITH]-(p) ; this helps duplicates be detected.

Note: The "ON CREATE SET" clause for the "Name" param ensures no overwriting

`add_metabolite_references`(*filename*)

Creates references for relations betweens Protein nodes and Metabolite nodes

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Warning: Unfortunately, Neo4J makes it really, really, really difficult to work with XML, and so, this time, a r.PubMed_ID list with the references could not be created. Nonetheless, I considered adding this useful.

`add_metabolites`(*filename*)

Creates "Metabolite" nodes based on XML files obtained from the HMDB website, adding some essential identifiers and external properties.

See also:

This way of working has been taken from William Lyon's Blog

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

`add_predicted_properties`(*filename*)

Adds properties to existing "Metabolite" nodes based on XML files obtained from the HMDB website. In this case, only properties labeled as <predicted_properties> are added.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Another option would have been to auto-add all the properties, and name them using RETURN "Predicted " + apoc.text.capitalizeAll(replace(kind, "_", " ")), value; however, this way we can select and not duplicate / overwrite values.

Todo: It would be nice to be able to distinguish between experimental and predicted properties

add_protein_associations(*filename*)

Creates "Protein" nodes based on XML files obtained from the HMDB website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Unlike the "add_protein" function, this creates Proteins based on info on the "Metabolite" files, not on the "Protein" files themselves. This could mean node duplication, but, hopefully, the MERGE by Accession will mean that this duplicates will be catched.

add_protein_properties(*filename*)

Adds some properties to existing "Protein" nodes based on XML files obtained from the HMDB website. In this case, properties will mostly relate to the protein itself.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: The "signal_regions" and the "transmembrane_regions" properties were left out because, after a preliminary search, they were mostly empty

add_proteins(*filename*)

Creates "Protein" nodes based on XML files obtained from the HMDB website.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: We are not creating "Gene" nodes (even though each protein comes from a given gene) because we believe not enough information is being given about them.

add_taxonomy(*filename*)

Creates "Taxonomy" nodes based on XML files obtained from the HMDB website. These represent the "kind" of metabolite we are dealing with (Family, etc)

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the XML file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: It only creates relationships in the Kingdom -> Super Class -> Class -> Subclass direction, and from any node -> Metabolite. This means that, if any member of the Kingdom -> Super Class -> Class -> Subclass is absent, the line will be broken; hopefully in that case a new metabolite will come in to rescue and settle the relation!

---

build_from_metabolite_file(*newfile*, *driver*)

A function able to build a portion of the HMDB database in graph format, provided that one "Metabolite" XML is supplied to it. This are downloaded separately from the website, as all the files that are not `hmdb_proteins.zip`, and can be presented either as the full file, or as a splitted version of it, with just one item per file (which is recommended due to memory limitations)

Parameters

- **newfile** (*str*) – The path of the XML file to import
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

build_from_protein_file(*newfile*, *driver*)

A function able to build a portion of the HMDB database in graph format, provided that one "Protein" XML is supplied to it. This are downloaded separately from the website, as `hmdb_proteins.zip`, and can be presented either as the full file, or as a splitted version of it, with just one item per file (which is recommended due to memory limitations)

Parameters

- **newfile** (*str*) – The path of the XML file to import
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

CanGraph.GraphifyHMDB.main module

A python module that leverages the functions present in the *build_database* module to recreate the HMDB database using a graph forma and Neo4J, and then provides an GraphML export file.

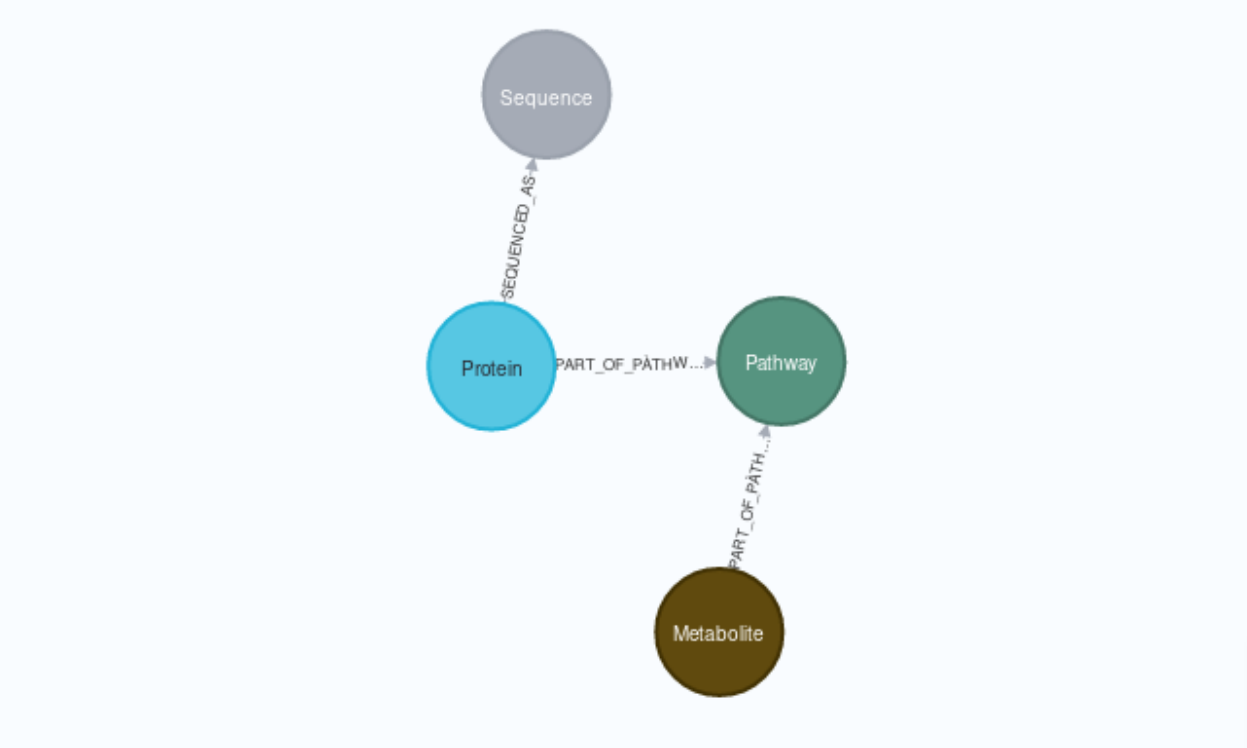
Please note that, to work, the functions here pre-suppose you have internet access, which will be used to download HMDB's XMLs under `./xmlfolder/` (please ensure you have read-write access there).

For more details on how to run this script, please consult the package's README

main()

The function that executes the code

3.5.4 CanGraph.GraphifySMPDB package



This script, created as part of my Master's Intenship at IARC, transitions the Small Molecule Pathway Database (a high quality database containing associations betweem metabolites, proteins and metabolomic pathways) to Neo4J format in an automated way, providing an export in GraphML format.

To run, it uses `alive_progress` to generate an interactive progress bar (that shows the script is still running through its most time-consuming parts) and the `neo4j` python driver. This requirements can be installed using: `pip install -r requirements.txt`.

To run the script itself, use:

```
python3 main.py neo4jadress databasename databasepassword
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **databasename**: the name of the database in use. If using the free version, there will only be one database per project (neo4j being the default name); if using the pro version, you can specify an alternate name here

- **databasepassword**: the passowrd for the **databasename** DataBase. Since the arguments are passed by BaSH onto python3, you might need to escape special characters

NOTE: The files will be downloaded to ./csvfolder, so please run the script somewhere you have read/write permissions

An archived version of this repository that takes into account the gitignored files can be created using: `git archive HEAD -o ${PWD##*/}.zip`

Important Notices on SMPDB

- Please ensure you have internet access, enough espace in your hard drive (around 5 GB) and read-write access in `./csvfolder`. The files needed to build the database will be stored there.
- Since the "Structure" files at smpdb.ca seemed to be super complicated to import, we decided against doing so. However, regarding the future, they shouldn't be overlooked, as they might include useful info
- The "PW ID" column from the "Pathways" table, and the "Pathway Subject" from the "Metabolite" tables may correlate (both start with PW). However, they seem to use different formats, so we have decided against including them in the Final Database

---

The package consists of the following modules:

CanGraph.GraphifySMPDB.build_database module

A python module that provides the necessary functions to transition the SMPDB database to graph format, either from scratch importing all the nodes (as showcased in *CanGraph.GraphifySMPDB.main*) or in a case-by-case basis, to annotate existing metabolites (as showcased in *CanGraph.main*).

add_metabolites(*filename*)

Adds "Metabolite" nodes to the database, according to individual CSVs present in the SMPDB website

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Some of the node's properties might be set to "null" (important in order to work with it)

Note: This database clearly differentiates Metabolites and Proteins, so no overlap is accounted for

---

add_pathways(*filename*)

Adds "Pathways" nodes to the database, according to individual CSVs present in the SMPDB website Since this is done after the creation of said pathways in the last step, this will most likely just annotate them.

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Todo: This file is really big. It could be divided into smaller ones.

`add_proteins`(*filename*)

Adds "Protein" nodes to the database, according to individual CSVs present in the SMPDB website

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Some of the node's properties might be set to "null" (important in order to work with it)

Note: This database clearly differentiates Metabolites and Proteins, so no overlap is accounted for

---

Todo: Why is the SMPDB_ID property called like that and not SMDB_ID?

Warning: Since no unique identifier was found, CREATE had to be used (instead of merge). This might create duplicates. which should be accounted for.

`add_sequence`(*seq_id*, *seq_name*, *seq_type*, *seq*, *seq_format='FASTA'*)

Adds "Pathways" nodes to the database, according to the sequences presented in FASTA files from the SMPDB website

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **seq_id** (*str*) – The UniProt Database Identifier for the sequence that is been imported
- **seq_name** (*str*) – The Name (i.e. FASTA header) of the Sequence that is been imported
- **seq_type** (*bool*) – The type of the sequence; can be either of ["DNA", "PROT"]
- **seq** (*str*) – The seuqnce that is been imported; a text chain of nucleotides or aminoacids, identified by their acronyms
- **seq_format** (*str*) – The format the sequence is provided under; default is "FASTA", but its optional

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

build_from_file(*filepath*, *Neo4JImportPath*, *driver*, *filetype*)

A function able to build a portion of the SMPDB in graph format, provided that one CSV is supplied to it. This CSVs are downloaded from the website, and can be presented either as the full file, or as a splitted version of it, with just one item per file (which is recommended due to memory limitations)

Since file title represents a different pathway, the function automatically picks up and import the relative pathway node.

Parameters

- **filepath** (*str*) – The path to the current file being imported
- **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **filetype** (*bool*) – The type of file being imported; one of ether ["Metabolite", "Protein"]- If the file is a FASTA sequence store, this will be auto-detected.

Returns

This function modifies the Neo4J Database as desired, but does not produce any particular return.

Note: Since this adds a ton of low-resolution nodes, maybe have this db run first?

CanGraph.GraphifySMPDB.main module

A python module that leverages the functions present in the *build_database* module to recreate the SMPDB database using a graph format and Neo4J, and then provides an GraphML export file.

Please note that, to work, the functions here pre-suppose you have internet access, which will be used to download HMDB's CSVs under `./csvfolder/` (please ensure you have read-write access there).

For more details on how to run this script, please consult the package's README

import_genomic_seqs()

Imports the `smpdb_gene.fasta` file from the SMPDB Database. The function assumes the file is availaible at `./csvfolder/smpdb_gene.fasta`

Parameters **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

import_metabolites(*filename*, *Neo4JImportPath*)

Imports "Metabolite" files from the SMPDB Database. The function assumes `filename` is availaible at `./csvfolder/smpdb_metabolites/`

Parameters

- **filename** (*str*) – The name of the CSV file that is being imported
- **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

import_pathways(*Neo4JImportPath*)

Imports the `smpdb_pathways.csv` file from the SMPDB Database. The function assumes the file is availaible at `./csvfolder/smpdb_gene.fasta`

Parameters **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

`import_proteic_seqs`()

Imports the `smpdb_protein.fasta` file from the SMPDB Database. The function assumes the file is availaible at `./csvfolder/smpdb_protein.fasta`

Parameters **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

`import_proteins`(*filename*, *Neo4JImportPath*)

Imports "Protein" files from the SMPDB Database. The function assumes `filename` is availaible at `./csvfolder/smpdb_proteins/`

Parameters

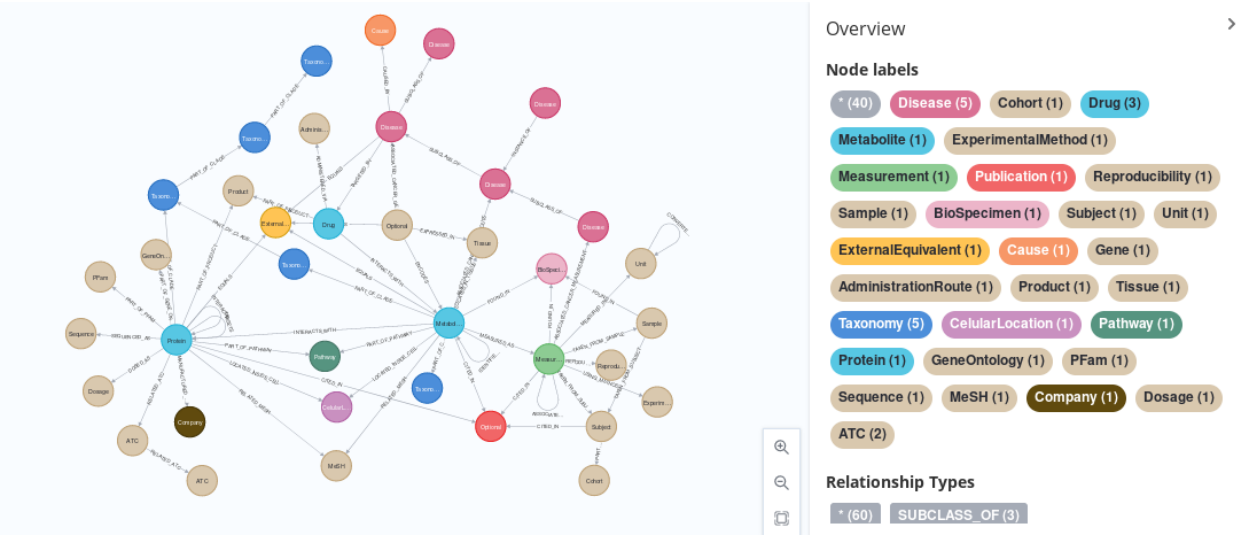
- **filename** (*str*) – The name of the CSV file that is being imported
- **Neo4JImportPath** (*str*) – The path from which Neo4J is importing data

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

`main`()

The function that executes the code

3.5.5 CanGraph.MeSHandMetaNetX package



This Git Project, created as part of my Master's Intenship at IARC, contains a series of scripts that pulls information from a series of **five** databases from their native format (XML, CSV, etc) into a common, GraphML format, using a shared schema that has been defined to minimize the number of repeated nodes and properties. This databases are:

- Exposome-Explorer: A hand-curated, high-quality database of associations between metabolites, food intakes and outakes and different diseases, specially cancers.
- Human Metabolome DataBase: An detailed, electronic database containing detailed information about small molecule metabolites found in the human body.

- DrugBank: A unique bioinformatics and cheminformatics resource that combines detailed drug data with comprehensive drug target information.
- Small Molecule Pathway Database: An interactive database containing more than 618 small molecule pathways found in humans, More than 70% of which are unique to this DB
- WikiData: The world's largest collaboratively generated collection of Open Data worldwide.

Each of then have their unique advantages and disadvantages (size, quality, etc) but they have been chosen to work together and help in identifying metabolites and their potential cancer associations at IARC.

With regards to the schema, it can be consulted in detail in the `new-schema.graphml` file, which can itself be opened in Neo4J by calling: `CALL apoc.import.graphml("new-schema.graphml", {useTypes:true, storeNodeIds:false, readLabels:True})` after placing it in your Neo4J's import directory (you can find it in the settings shown after starting the server with `sudo neo4j start`). It consists of a simplification of all the nodes present on the `old-schema.graphml` file (which itself represents the five different schemas that our five databases natively presented), arrived at by merging nodes and changing relationship names so that they are unique (and, thus, more actionable). One property, `LabelName` has been added as a dummy name to generate the image you can see in the header.

This repo contains two kind of scripts: first, some `build_database.py` scripts, which contain the information to re-build the databases in the common format from scratch, and are located in subsequent subfolders named after the database they come from (more info can be consulted on them on their respective READMEs) and a common `main.py` script, which can be used to query for sub-networks based solely on info presented on a **sample_input.csv** database of identified compounds which we would like to annotate.

Intallation

To use this script, you should first clone it into your personal computer. The easiest way to do this is to git clone the repo:

1. Install git (if not already installed) and other requirements. On linux: `sudo apt install git curl`
2. Clone the repo: `git clone https://codeberg.org/FlyingFlamingo/graphify-databases`
3. Step into the directory `cd graphify-databases`

Once the project has been installed, you **must** run `setup.py`, a preparation script that guides you through the process of installing all five databases on your computer, so that then we can correctly process them and generate the sub-networks. You should also install the required python modules and run the setup script:

4. PIP install all dependencies: `pip install -r requirements.txt`
5. Run the setup script: `python3 setup.py`

Once this has been done, you are ready to start using the main script!

NOTE: If you do not wish to use git, you can manually download the repo by clicking here

Usage

To generate this sub-networks (the original idea of the project) you should run:

```
python3 main.py neo4jadress databaseusername databasepassword databasefolder inputfile
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **username**: the username for your neo4j instance. Remember, the default is neo4j

- **password**: the password for your database. Since the arguments are passed by BaSH onto python3, you might need to escape special characters
- **databasefolder**: The folder indicated to `setup.py` as the one where your databases will be stored
- **inputfile**: The location of the CSV file in which the program will search for metabolites. This file should be a Comma-Separated file, with the following format: `MonoisotopicMass, SMILES, InChIKey, Name, InChI, Identifier, ChEBI`

NOTE: When committing to the repo, try to use GitMojis to illustrate your commit :p

Important Notices

- Some databases are auto-integrated based on their URLs. This URLs, as well as those of existing dependencies, may change over time. Please make sure to have them updated in case you want to run the latest version of the databases
- We have made our best efforts to make the script as multi-platform as possible; however, the script has been developed with Linux in mind, and you may need to install additional packages if you want to run it on Windows or MacOS. Please, check the `dependencies` section for more info

Dependencies

This python package has the following known dependencies:

Package	Description	Order to install
Python 3.8	the python programming language	`sudo apt install python3`
cURL	command line tool for transferring data from URLs	`sudo apt install curl`
neo4j	a graph dbms	`python3 -c'import setup; setup.setup_neo4j("neo4j", True)'`

Alternatively, as a one-liner: `sudo apt install python3 curl; python3 -c'import setup; setup.setup_neo4j("neo4j", True)`

The package consists of the following modules:

CanGraph.MeSHandMetaNetX.build_database module

A python module that provides the necessary functions to transition the MetaNetX database (and related MeSH terms and KEGG IDs) to graph format, either from scratch importing all the nodes (as showcased in *CanGraph.MeSHandMetaNetX.main*) or in a case-by-case basis, to annotate existing metabolites (as showcased in *CanGraph.main*).

Note: You may notice some functions here present the `**kwargs` arguments option. This is in order to make the functions compatible with the *CanGraph.miscelaneous.manage_transaction* function, which might send back a variable number of arguments (although technically it could work without the `**kwargs` option)

---

add_chem_isom(*filename*)

A CYPHER query that loads the *chem_isom.tsv* file availaible at the MetaNetX site, using a graph format.

Parameters **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: For performance, it is recommended to split the file in 1 subfile for each row in the DataBase

add_chem_prop(*filename*)

A CYPHER query that loads the *chem_prop.tsv* file availaible at the MetaNetX site, using a graph format.

Parameters **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: For performance, it is recommended to split the file in 1 subfile for each row in the DataBase

add_chem_xref(*filename*)

A CYPHER query that loads the *chem_xref.tsv* file availaible at the MetaNetX site, using a graph format.

Parameters **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: For performance, it is recommended to split the file in 1 subfile for each row in the DataBase

add_comp_prop(*filename*)

A CYPHER query that loads the *comp_prop.tsv* file availaible at the MetaNetX site, using a graph format.

Parameters **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: For performance, it is recommended to split the file in 1 subfile for each row in the DataBase

add_comp_xref(*filename*)

A CYPHER query that loads the *comp_xref.tsv* file availaible at the MetaNetX site, using a graph format.

Parameters **filename** (*str*) – The name of the CSV file that is being imported

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: For performance, it is recommended to split the file in 1 subfile for each row in the DataBase

Note: Some identifiers present the CL/cl prefix. Since I could not find what this prefix refers to, and since it only pertains to one single MetaNetX ID, we did not take them into account

Note: The "description" field in the DataBase is ignored, since it seems to be quite similar, but less useful, than the "name" field from comp_prop, which is more coherent with our pre-existing schema

`add_mesh_by_name()`

A function that adds some MeSH nodes to any existing nodes, based on their Name property. Only currently active MeSH_IDs are parsed

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: `neo4j.Session.run`

Return type str

Note: Only exact matches work here, which is not ideal.

Note: Be careful when writing CYPHER commands for the driver: sometimes, " != ' !!!

Changed in version 1.0: Reverted the filtering to old version in order to make the search more specific

`add_pept()`

A CYPHER query that all the protein availaible at the MetaNetX site, using a graph format and SPARQL.

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: `neo4j.Session.run`

Return type str

Note: SPARQL was only used here because, unlike with the other files, there is no download available; also, given there are few proteins, Neo4J is able to process it without running out of memory (unlike what happened with the other fields)

Note: This is an **autocommit transaction**. This means that, in order to not keep data in memory (and make running it with a huge amount of data) more efficient, you will need to add `` `:auto ` `` when calling it from the Neo4J browser, or call it as `` `session.run( clean_database() )` `` from the driver.

`add_prefixes()`

Add some prefixes necessary for all MetaNetX queries to work. This are kept together since adding extra prefixes does not increase computation time

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: `neo4j.Session.run`

Return type str

build_from_file(*filename*, *driver*)

A function able to build a portion of the MetaNetX database in graph format, provided that one MetaNetX CSV is supplied to it. This CSVs are downloaded from the website, and can be presented either as the full file, or as a splitted version of it, with just one item per file (which is recommended due to memory limitations). If you want all the database to be imported, you should run this function with all the CSVs that form it, as portrayed in the *main* module

Parameters

- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **filename** (*str*) – The name of the CSV file that is being imported

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

find_metabolites_related_to_mesh(*mesh_id*)

A function that finds Metabolites related to a given MeSH ID, on the MeSH DataBase

Parameters **mesh_id** (*str*) – The MeSH_ID of the thing for which we want to find related proteins

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: neo4j.Session.run

Return type str

Note: This is intended to be run as a execute_read, only returning synonyms present in the DB. No modifications will be applied.

Note: Could be turned into a read query by substituting `mesh_id` with `' + n.MeSH_ID + '`

---

find_protein_data_in_metanetx()

A SPARQL function that annotates Protein nodes in an exiting Neo4J database by using the information provided by MetaNetX

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: neo4j.Session.run

Return type str

Note: This function is partly a duplicate of *self.find_protein_interactions_in_metanetx()*, which was split to prevent timeouts

---

find_protein_interactions_in_metanetx()

A SPARQL function that finds the Metabolites a given Protein (based on its UniProt_ID) interacts with, using MetaNetX.

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: neo4j.Session.run

Return type str

Note: We are not using peptXref: since all proteins in MetaNetX come from UniProt, there is no use here

---

find_synonyms_in_cts(*fromIdentifier*, *toIdentifier*, *searchTerm*)

Finds synonyms for a given metabolite in CTS, The Chemical Translation Service

Parameters

- **fromIdentifier** (*str*) – The name of the database from which we want the conversion
- **toIdentifier** (*str*) – The name of the database to which we want the conversion
- **searchTerm** (*str*) – The search termm, which should be an ID of database: fromIdentifier

Returns The requested synonym

Return type str

Note: Please, be sure to use a database name that is in compliance with those specified in CTS itself; if you dont, this function will fail with a 500 error

---

Note: To prevent random downtimes from crashing the function, any one URL will be tried at least 5 times before crashing (see: StackOverflow #9446387

---

get_identifiers(*from_sparql=False*, ***kwargs*)

Part of a CYPHER query that processes the outcome from a SPARQL query that searches for information on MetaNetX It takes an original metabolite (n) and a row variable, which should have columns named external_identifier, cross_refference, InChIKey, InChI, SMILES, Formula and Mass with the adequated format; it is basically a code-reuser, not intended to be used separately.

Parameters

- **from_sparql** (*bool*) – A True/False param defining whether the identifiers are being parsed from a SPARQL query; default is False (i.e. imported from file)
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: neo4j.Session.run

Return type str

Note: All HMDB matches might create a Metabolite without CHEBI_ID or CAS_Number, which would violate our schema. This will be later on accounted for.

Note: Some keys, such as VMH_ID, are not merged into their own node, but rather added to an existing one. This is because this do not prevously exist in our Schema, and might be changed in the future.

Note: We dont care about overwriting InChI and InChIKey because they are necessarily unique; the same is true for Mass and Formula, as they are not all that important. However, for HMDB ID and others, we will take care not to overwrite, which could mess up the DB

`get_kegg_pathways_for_metabolites()`

A function that finds the Pathways a given Metabolite (based on its Kegg_ID) is a part of, using KEGG. This uses genome.jp's dbget web service, since I honestly could not find a way to use KEGG's SPARQL service (https://www.genome.jp/linkdb/linkdb_rdf.html) for that.

See also:

Another possibility could be using Kegg's Rest API

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: `neo4j.Session.run`

Return type str

`read_synonyms_in_metanetx(query_type, query, **kwargs)`

A SPARQL function that finds synonyms for metabolites, proteins or drugs based on a given *query*, using MetaNetX. At the same time, it is able to annotate them a bit, adding Name, InChI, InChIKey, SMILES, Formula, Mass, some External IDs, and finding whether the metabolite in question has any known isomers, anootating if so.

Parameters

- **query_type** (*str*) – The type of query that is being searched for. One of ["Name","KEGG_ID","ChEBI_ID","HMDB_ID","InChI","InChIKey"]
- **query** (*str*) – The query we are searching for; must be of type `query_type`
- ****kwargs** – Any number of arbitrary keyword arguments

Returns:

Raises `ValueError` – If the query type is not one of those accepted by the function

---

Note: This is intended to be run as a execute_read, only returning synonyms present in the DB. No modifications will be applied.

`write_synonyms_in_metanetx(query, **kwargs)`

A SPARQL function that finds synonyms for metabolites, proteins or drugs in an existing Neo4J database, using MetaNetX. At the same time, it is able to annotate them a bit, adding Name, InChI, InChIKey, SMILES, Formula, Mass, some External IDs, and finding whether the metabolite in question has any known isomers, anootating if so.

Parameters

- **query** (*str*) – The type of query that is being searched for. One of ["Name","KEGG_ID","ChEBI_ID","HMDB_ID","InChI","InChIKey"].
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: `neo4j.Session.run`

Return type str

Raises `ValueError` – If the query type is not one of those accepted by the function

---

Note: This is intended to be run as a manage_transaction, modifying the existing database.

CanGraph.MeSHandMetaNetX.main module

A python module that leverages the functions present in the *build_database* module to recreate the MetaNetX database using a graph format and Neo4J, and then provides an GraphML export file. It also annotates related MeSH_IDs and KEGG Pathway IDs

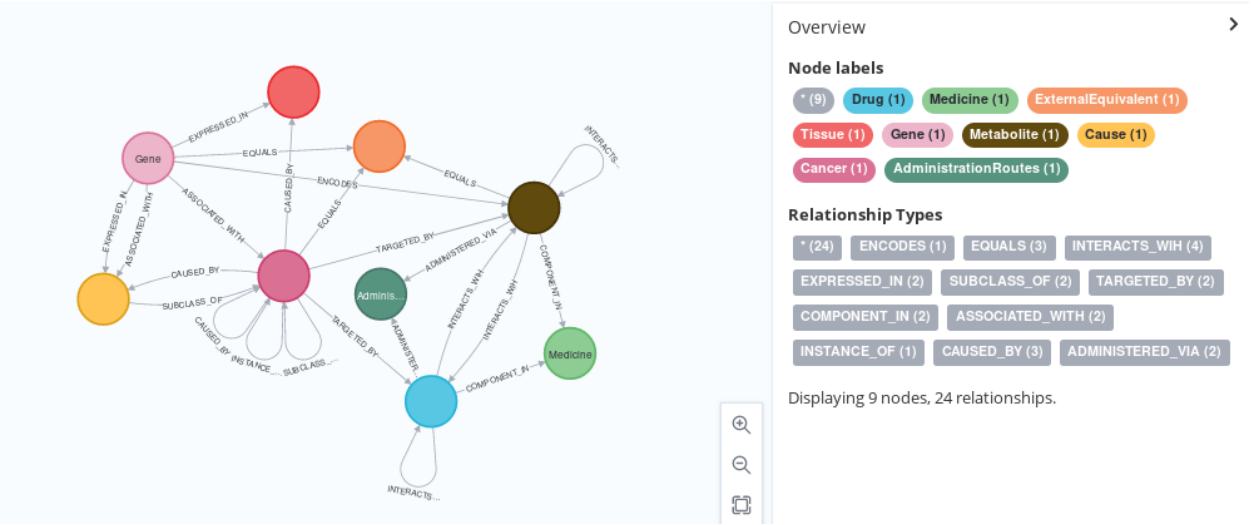
Please note that, to work, the functions here pre-suppose you have internet access, which will be used to download MetaNetX's TSVs under a folder provided as `sys.argv[4]`. (please ensure you have read-write access there) and query some web SPARQL and REST web services.

For more details on how to run this script, please consult the package's README

main()

The function that executes the code

3.5.6 CanGraph.QueryWikidata package



This script, created as part of my Master's Intenship at IARC, imports nodes from the WikiData SPARQL Service, creating a high-quality representation of the data therein. Although wikidata is manually curated using the Wiki principles, some publications have found it might be a good source of information for life sciences, specially due to the breadth of information it contains. It also provides an export in GraphML format.

To run, it uses `alive_progress` to generate an interactive progress bar (that shows the script is still running through its most time-consuming parts) and the `neo4j` python driver. This requirements can be installed using: `pip install -r requirements.txt`.

To run the script itself, use:

```
python3 build_database.py neo4jadress username databasepassword
```

where:

- **neo4jadress**: is the URL of the database, in neo4j:// or bolt:// format
- **username**: the username for your neo4j instance. Remember, the default is neo4j
- **password**: the passowrd for your database. Since the arguments are passed by BaSH onto python3, you might need to escape special characters

Please note that there are two kinds of functions in the associated code: those that use python f-strings, which themselves contain text that *cannot* be directly copied into Neo4J (for instance, double brackets have to be turned into simple

brackets) and normal multi-line strings, which can. This is because f-strings allow for variable customization, while normal strings dont.

An archived version of this repository that takes into account the gitignored files can be created using: `git archive HEAD -o ${PWD##*/}.zip`

Finally, please node that the general philosophy and approach of the queries have been taken from Towards Data Science, a genuinely useful web site.

Important Notices on WikiData

- Please ensure you have internet access, which will be used to connect to Wikidata's SPAQL endpoint and gather the necessary info.
- As Neo4J can run out of "Java Heap Space" if the number of nodes/properties to add is too high, the script has been divided in order to minimize said number: for instance, only nodes with a `wikidata_id` ending in a given number from 0 to 9 are processed at a time. This does not decrease performance, since these nodes would have been processed nontheless, but makes the script more reliable.
- What does impact performance, however, is having different functions for adding cancers, drugs, metabolites, etc, instead of having just one match for each created cancer node. This makes WikiData have to process more queries that are less heavy, which makes it less likely to time-out, but causes the script to run more slowly.
- The Neo4J server presents a somewhat unstable connection that is sometimes difficult to keep alive, as it tends to be killed by the system when you so much as look at it wrong. To prevent this from happening, you are encouraged to assign a high-priority to the server's process by using the `nice` or `renice` commands in Linux (note that the process will be called "Java", not "Neo4J")
- Another measure taken to prevent Neo4J's unreliability from stopping the script is the `misc.manage_transaction` function, which insists a given number of times until either the problem is fixed or the error persists. This is because Neo4J tends to: random disconnects, run out of java heap space, explode… and WikiData tends to give server errors, have downtimes during the **14+ hours** the script takes to run, etc.
- The data present in the "graph.graphml" file comes from WikiData, and was provided by this service free of charge and of royalties under the permissive CC-0 license.

The package consists of the following modules:

CanGraph.QueryWikidata.build_database module

A python module that provides the necessary functions to transition selected parts of the Wikidata database to graph format, either from scratch importing all the nodes (as showcased in *CanGraph.QueryWikidata.main*) or in a case-by-case basis, to annotate existing metabolites (as showcased in *CanGraph.main*).

Note: You may notice some functions here present the `**kwargs` arguments option. This is in order to make the functions compatible with the *CanGraph.miscelaneous.manage_transaction* function, which might send back a variable number of arguments (although technically it could work without the `**kwargs` option)

---

add_causes(*number*, ***kwargs*)

Creates drug nodes related with each of the "Cancer" nodes already on the database

Parameters

- **number** (*int*) – From 0 to 9, the number under which the WikiData_IDs to process should ends. This allows us tho divide the work, although its not very elegant.

- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: Here, there is no need to force c.WikiData_ID to not be null or "" because it will already be = number (and, thus, exist)

---

add_disease_info(*number*, ***kwargs*)

Adds info to "Disease" nodes for which its WikiData_ID ends in a given number. This way, only some of the nodes are targeted, and the Java Virtual Machine does not run out of memory

Parameters

- **number** (*int*) – From 0 to 9, the number under which the WikiData_IDs to process should ends. This allows us tho divide the work, although its not very elegant.
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: Here, there is no need to force c.WikiData_ID to not be null or "" because it will already be = number (and, thus, exist)

---

add_drug_external_ids(*query='Wikidata_ID'*, ***kwargs*)

Adds some external IDs to any "Drug" nodes already present on the database. Since the PDB information had too much values which caused triple duplicates that overcharged the system, they were intentionally left out.

Parameters

- **query** (*str*) – One of ["DrugBank_ID","WikiData_ID"], a way to identify the nodes for which external IDs will be added.
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: We are forcing c.WikiData_ID to not be null or "". This is not necessary if we are just building the wikidata database, because there will always be a WikiData_ID, but it is useful in the rest of the cases

---

add_drugs(*number*, ***kwargs*)

Creates drug nodes related with each of the "Cancer" nodes already on the database

Parameters

- **number** (*int*) – From 0 to 9, the number under which the WikiData_IDs to process should ends. This allows us tho divide the work, although its not very elegant.
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: Here, there is no need to force c.WikiData_ID to not be null or "" because it will already be = `number` (and, thus, exist)

`add_gene_info()`

A Cypher Query that adds some external IDs and properties to "Gene" nodes already existing on the database. This query forces the genes to have a "found_in_taxon:homo_sapiens" label. This means that any non-human genes will not be annotated (.. TODO:: delete those)

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: Genomic Start and ends keep just the 2nd position, as reported in wikidata

---

Note: We are forcing c.WikiData_ID to not be null or "". This is not necessary if we are just building the wikidata database, because there will always be a WikiData_ID, but it is useful in the rest of the cases

---

Todo: Might include P684 "Orthologues" for more info (it crashed java)

`add_genes(number, **kwargs)`

Creates gene nodes related with each of the "Cancer" nodes already on the database

Parameters

- **number** (*int*) – From 0 to 9, the number under which the WikiData_IDs to process should ends. This allows us tho divide the work, although its not very elegant.
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: Here, there is no need to force c.WikiData_ID to not be null or "" because it will already be = `number` (and, thus, exist)

`add_metabolite_info(query='ChEBI_ID', **kwargs)`

A Cypher Query that adds some external IDs and properties to "Metabolite" nodes already existing on the database. Two kind of metabolites exist: those that are encoded by a given gene, and those that interact with a given drug. Both are adressed here, since they are similar, and, most likely, instances of proteins.

- This function forces all metabolites to have a "found_in_taxon:human" target
- The metabolites are not forced to be proteins, but if they are, this is kept in the "instance_of" record

Parameters

- **query** (*str*) – One of ["DrugBank_ID","WikiData_ID"], a way to identify the nodes for which external IDs will be added; default is "WikiData_ID"
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Todo: Might include P527 "has part or parts" for more info (it crashed java)

Note: We are forcing c.WikiData_ID to not be null or "". This is not necessary if we are just building the wikidata database, because there will always be a WikiData_ID, but it is useful in the rest of the cases

---

add_more_drug_info(*query='WikiData_ID'*, ***kwargs*)

Creates some nodes that are related with each of the "Drug" nodes already existing on the database: routes of administration, targeted metabolites and approved drugs that tehy are been used in

Parameters

- **query** (*str*) – One of ["DrugBank_ID","WikiData_ID"], a way to identify the nodes for which external IDs will be added; default is "WikiData_ID"
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Todo: ADD ROLE to metabolite interactions

Note: This transaction has been separated in order to keep response times low

---

Note: We are forcing c.WikiData_ID to not be null or "". This is not necessary if we are just building the wikidata database, because there will always be a WikiData_ID, but it is useful in the rest of the cases

---

add_toomuch_metabolite_info()

A function that adds loads of info to existing "Metabolite" nodes. This was left out, first because it might be too much information, (specially when it is already availaible by clicking the "url" field), and because, due to it been so much, it crashes the JVM.

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

`add_wikidata_and_mesh_by_name()`

A function that adds some MeSH nodes and WikiData_IDs to existing nodes, based on their Wikipedia Article Title.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

`add_yet_more_drug_info(query='WikiData_ID', **kwargs)`

Creates some nodes that are related with each of the "Drug" nodes already existing on the database: routes of administration, targeted metabolites and approved drugs that tehy are been used in

Parameters

- **query** (*str*) – One of ["DrugBank_ID","WikiData_ID"], a way to identify the nodes for which external IDs will be added; default is "WikiData_ID"
- ****kwargs** – Any number of arbitrary keyword arguments

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

`find_instance_of_disease()`

A Neo4J Cypher Statment that queries wikidata for instances of "Disease" nodes already present on the Database. Since these are expected to only affect humans, this subclasses should also, only affect humans

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: We are forcing c.WikiData_ID to not be null or "". This is not necessary if we are just building the wikidata database, because there will always be a WikiData_ID, but it is useful in the rest of the cases

---

`find_subclass_of_disease()`

A Neo4J Cypher Statment that queries wikidata for subclasses of "Disease" nodes already present on the Database. Since these are expected to only affect humans, this subclasses should also, only affect humans

Returns A CYPHER query that modifies the DB according to the CYPHER statement contained in the function.

Return type str

Note: We are forcing c.WikiData_ID to not be null or "". This is not necessary if we are just building the wikidata database, because there will always be a WikiData_ID, but it is useful in the rest of the cases

---

`initial_cancer_discovery()`

A Neo4J Cypher Statment that queries wikidata for Human Cancers. Since using the "afflicts:human" tag didnt have much use here, I used a simple workaround: Query wikidata for all humans, and, among them, find all of this for which their cause of death was a subclass of "Cancer" (Q12078). Unfortunaltely, some of them were diagnosed "Cancer" (Q12078), which is too general, so I removed it.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

CanGraph.QueryWikidata.main module

A python module that leverages the functions present in the *build_database* module to recreate selected parts of the the Wikidata database using a graph format and Neo4J, and then provides an GraphML export file.

Please note that, to work, the functions here pre-suppose you have internet access, which will be used to access Wikidata's SPAQL endpoint and write info to the Neo4J database

For more details on how to run this script, please consult the package's README

main()

The function that executes the code

3.5.7 Install Apptainer

```
#!/usr/bin/env bash

# SPDX-FileCopyrightText: 2022 Pablo Marcos <software@loreak.org>
#
# SPDX-License-Identifier: MIT

# ************************ #
# Install APPTainer
# ************************ #

# This is the a simplified script that installs AppTainer v1.1.2
# on any debian-based OS (basically, one using BaSH and APT), using
# and installing Go v1.19 and the GoLangCI v1.43.0

# This script was taken from:
# https://github.com/apptainer/apptainer/blob/main/INSTALL.md

# Another option is to use a custom repo; see:
# https://docs.sylabs.io/guides/3.0/user-guide/installation.html
    #install-the-debian-ubuntu-package-using-apt

# ************************ #

# First, we update the repos and install some dependencies
sudo apt-get update && \
sudo apt-get install -y build-essential libseccomp-dev pkg-config \
    uidmap squashfs-tools squashfuse fuse2fs fuse-overlayfs fakeroot \
    cryptsetup curl wget git

# Then, we void the Go Folder and re-install the desired Go version
sudo rm -r /usr/local/go
export VERSION=1.19 OS=linux ARCH=amd64
wget -O /tmp/go${VERSION}.${OS}-${ARCH}.tar.gz \
```

(continues on next page)

(continued from previous page)

```
    https://dl.google.com/go/go${VERSION}.${OS}-${ARCH}.tar.gz && \
sudo tar -C /usr/local -xzf /tmp/go${VERSION}.${OS}-${ARCH}.tar.gz

# We appropriate the env vars accordingly using source and bashrc
echo 'export GOPATH=${HOME}/go' >> ~/.bashrc && \
echo 'export PATH=/usr/local/go/bin:${PATH}:${GOPATH}/bin' >> ~/.bashrc && \
source ~/.bashrc

# We will also need to install the GoLangCI
curl -sfL https://install.goreleaser.com/github.com/golangci/golangci-lint.sh |
sh -s -- -b $(go env GOPATH)/bin v1.43.0

# Then, we clone the git repo for APPTainer and checkout to the v3.6.3 branch
mkdir -p ${GOPATH}/src/github.com/apptainer && \
cd ${GOPATH}/src/github.com/apptainer && \
rm -rf apptainer/
git clone https://github.com/apptainer/apptainer.git && \
cd apptainer
git checkout v1.1.2

# Finally, we install apptainer using the included files
./mconfig && \
cd ./builddir && \
make && \
sudo make install

# And, we can check it
apptainer --version
```

``

3.6 CanGraph.deploy module

A python module that simplifies deploying the different formats the program can present itself as:

- A web documentation that is made using sphinx
- A PDF manual likewise made, also using LaTeX
- A git repo where the program can be accessed and version-tagged
- And, in the future… a singularity container!

3.6.1 CanGraph.deploy Usage

To use this module:

```
usage: python3 deploy.py [-h] [-m] [-d] [-w] [-p]
```

Named Arguments

-m, --main	deploy code to your dev branch
-d, --dev	deploy code to your main branch
-w, --web	deploys the documentation to the 'pages' web site, depending on which is activated
-p, --pdf	generates a PDF version of the sphinx manual, and saves it to the repo

Note: For this program to work, the Git environment **has to be set up first**. You can ensure this by using: *CanGraph.setup.setup_git*

3.6.2 CanGraph.deploy Functions

This module is comprised of:

args_parser()

Parses the command line arguments into a more usable form, providing help and more

Returns A dictionary of the different possible options for the program as keys, specifying their set value. If no command-line arguments are provided, the help message is shown and the program exits.

Return type argparse.ArgumentParser

Note: Note that, in Google Docstrings, if you want a multi-line `Returns` comment, you have to start it in a different line :(

Note: The return **must** be of type `argparse.ArgumentParser` for the `argparse` directive to work and autogen docs

deploy_code(*branch='dev'*)

Deploys code from a given branch to the corresponding remote.

Parameters **branch** (*str*) – The name of the branch of the **local** git repo that we want to deploy

Note: Normally, the `pages` branch should be published using *deploy_webdocs*, which in theory would be weird to publish without updating the docs first

deploy_pdf_manual(*docs_folder='./docs/'*, *work_dir='.'*, *manual_location='./CanGraph_Manual.pdf'*, *prechecks_done=False*, *custom_domain=None*)

Parses the command line arguments into a more usable form, providing help and more

Generates the PDF docs guide, and publishes it in `manual_location`

Parameters

- **docs_folder** (*str*) – The path to sphinx's docs folder, where the tests will be run; by default `./docs/`
- **work_dir** (*str*) – The current Working Directory; by default
- **prechecks_done** (*bool*) – Whether the prechecks present in *~CanGraph.deploy.make_sphinx_prechecks* have already been made
- **custom_domain** (*str*) – A custom domain to deploy de docs to.
- **manual_location** (*str*) – The location (**including filename**) of the finalised PDF manual, relative to the location of the script

Returns Whether the prechecks have already been done; always True if the function is run

Return type bool

deploy_webdocs(*docs_folder='./docs/'*, *work_dir='.'*, *prechecks_done=False*, *custom_domain=None*)

Generates the HTML web docs, and publishes it both to Github and Codeberg pages

Parameters

- **docs_folder** (*str*) – The path to sphinx's docs folder, where the tests will be run; by default `./docs/`
- **work_dir** (*str*) – The current Working Directory; by default
- **prechecks_done** (*bool*) – Whether the prechecks present in *~CanGraph.deploy.make_sphinx_prechecks* have already been made
- **custom_domain** (*str*) – A custom domain to deploy de docs to.

Returns Whether the prechecks have already been done; always True if the function is run

Return type bool

Note: For `custom_domain` to work, please configure your DNS records apparently

---

Note: `modules.rst` is not removed, but it is correctly ignored in conf.py

---

git_push(*path_to_repo*, *remote_names*, *commit_message*, *force=False*)

Pushes the current repo's state and current branch to a remote git repository

Parameters

- **path_to_repo** (*str*) – The path to the local `.git` folder
- **remote_names** (*list or str*) – The names of the remote to which we want to commit, which must be previously configured (see *CanGraph.setup.setup_git*). e.g.: ["github", "codeberg"]
- **commit_message** (*str*) – The Git Commit Message for the current repo's state
- **force** (*bool*) – Whether to force the commit (necessary if you are resetting the HEAD)

Note: `gitpython` is not good at managing complex commit messages (i.e. those with a Subject and a Body). If you want to add one of those, please, use `\n` as the separator; the function will take care of the rest

See also:

The approach taken hare was inspired by StackOverflow #41836988

`main`()

The function that executes the code

`make_sphinx_prechecks`(*docs_folder='./docs/'*, *work_dir='.'*, *gen_apidocs=False*)

Generates sphinx api-docs for automatic documentation and uses `make linkcheck`` to check for broken links

Parameters

- **gen_apidocs** (*bool*) – Whether to re-generate the API docs. Default is `False` since we use MD (we dont want RST files)
- **docs_folder** (*str*) – The path to sphinx's docs folder, where the tests will be run; by default `./docs/`
- **work_dir** (*str*) – The current Working Directory; by default `.`

3.7 CanGraph.main module

A python module that leverages the functions present in the *miscelaneous* module and all other subpackages to annotate metabolites using a graph format and Neo4J, and then provides an GraphML export file.

3.7.1 CanGraph.main Usage

To use this module:

A python utility to study and analyse cancer-associated metabolites using knowledge graphs

```
usage: python3 main.py [-h] [-c] [-n] [-s] [-w] [-i] --query QUERY
                       [--dbfolder DBFOLDER] [--results RESULTS]
                       [--adress ADRESS] [--username USERNAME]
                       [--password PASSWORD]
```

Named Arguments

-c, --check_args	Checks if the rest of the arguments are OK, then exits
-n, --noindex	Runs the program checking each file one-by-one, instead of using a JSON index
-s, --similarity	Deactivates the import of information based on Structural Similarity.This might dramatically increase processing time; default is True.
-w, --webdbs	Activates import of information based on web databases.This might dramatically increase processing time; default is True.
-i, --interactive	tells the script if it wants interaction from the user and more information shown to them; similar to –verbose

--query	The location of the CSV file in which the program will search for metabolites
--dbfolder	The folder indicated to `setup.py` as the one where your databases will be stored; default is `./DataBases`
--results	The folder where the resulting GraphML exports will be stored; default is `./Results`
--adress	the URL of the database, in neo4j:// or bolt:// format
--username	the username of the neo4j database in use
--password	the password for the neo4j database in use. NOTE: Since passed through bash, you may need to escape some chars

You may find more info in the package's README.

Note: For this program to work, the Git environment **has to be set up first**. You can ensure this by using: *CanGraph.setup.setup_git*

3.7.2 CanGraph.main Functions

This module is comprised of:

add_mesh_and_metanetx(*driver*)

Add MeSH Term IDs, Synonym relations and Protein interactions to existing nodes using MeSH and MetaNetX Also, adds Kegg Pathway IDs

Parameters **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

annotate_using_wikidata(*driver*)

Once we finish the search, we annotate the nodes added to the database using WikiData

Parameters **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

Todo: When fixing queries, fix the main subscript also

args_parser()

Parses the command line arguments into a more usable form, providing help and more

Returns A dictionary of the different possible options for the program as keys, specifying their set value. If no command-line arguments are provided, the help message is shown and the program exits.

Return type argparse.ArgumentParser

Note: Note that, in Google Docstrings, if you want a multi-line `Returns` comment, you have to start it in a different line :(

Note: The return **must** be of type `argparse.ArgumentParser` for the `argparse` directive to work and autogen docs

Note: By using `argparse.const` instead of `argparse.default`, the check_file function will check "" (the current dir, always exists) if the arg is not provided, not breaking the function; if it is, it checks it.

`build_from_file`(*filepath*, *Neo4JImportPath*, *driver*)

Imports a given metabolite from a sigle-metabolite containing file by checking its type and calling the appropriate import functions.

Parameters

- **filepath** (*str*) – The path to the file in which will be imported
- **Neo4JImportPath** (*str*) – The path which Neo4J will use to import data
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns This function does not provide a particular return, but rather imports the requested file

Note: The `filepath` may be absolute or relative, but it is transformed to a relative `relpath` in order to remove possible influence of higher-name folders in the import type selection. This is also why the condition is stated as a big "if/elif/else" instead of a series of "ifs"

`find_reasons_to_import_all_files`(*filepath*, *similarity*, *chebi_ids*, *names*, *hmdb_ids*, *inchis*, *mesh_ids*)

Finds reasons to import a metabolite given a candidate filepath **with one metabolite per file** and a series of lists containing all synonyms of the values considered reasons for import

Parameters

- **filepath** (*str*) – The path to the file in which we will search for reasons to import
- **similarity** (*bool*) – Whether to use similarity as a measure to import or not
- **chebi_ids** (*list*) – A list of all the ChEBI_ID which are considered a reason to import
- **names** (*list*) – A list of all the Name which are considered a reason to import
- **hmdb_ids** (*list*) – A list of all the HMDB_ID which are considered a reason to import
- **inchis** (*list*) – A list of all the InChI which are considered a reason to import
- **mesh_ids** (*list*) – A list of all the MeSH_ID which are considered a reason to import

Returns A list of the methods that turned out to be valid for import, such as Name, ChEBI_ID…

Return type list

`find_reasons_to_import_inchi`(*query*, *subject*)

Takes two chains of text and finds if the `query` is present in the `subject`, or if there are molecules common between them with at least 95% similarity

Parameters

- **query** (*str or list*) – A string or list of strings describing valid InChI(s)
- **subject** (*str*) – A valid InChI

Returns A dict with each query as a key and the reason to import it as value, if there is one.

Return type dict

See also:

This approach was taken from Chemistry StackExchange #82144

Note: Since this is a one-to-one comparison, subject and query can be used interchangeably; however, bear in mind that only the query can be provided as a list

---

import_based_on_all_files(*all_files*, *Neo4JImportPath*, *driver*, *similarity*, *chebi_ids*, *names*, *hmdb_ids*, *inchis*, *mesh_ids*)

A function that searches inside a series of lists, provided as arguments, and imports the metabolites matching those present in them iterating over a list of files which may contain relevant information to be imported

Parameters

- **all_files** (*list*) – A list of all the posible files where we want to look for info
- **Neo4JImportPath** (*str*) – The path which Neo4J will use to import data
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **similarity** (*bool*) – Whether to use similarity as a measure to import or not
- **chebi_ids** (*list*) – A list of all the ChEBI_ID which are considered a reason to import
- **names** (*list*) – A list of all the Name which are considered a reason to import
- **hmdb_ids** (*list*) – A list of all the HMDB_ID which are considered a reason to import
- **inchis** (*list*) – A list of all the InChI which are considered a reason to import
- **mesh_ids** (*list*) – A list of all the MeSH_ID which are considered a reason to import

import_based_on_index(*databasefolder*, *Neo4JImportPath*, *driver*, *similarity*, *chebi_ids*, *names*, *hmdb_ids*, *inchis*, *mesh_ids*)

A function that searches inside a series of lists, provided as arguments, and imports the metabolites matching those present in them using a JSON file to map the bits of the databases where the relevant information lies

Parameters

- **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found There *must* be an index.json file located in `databasefolder`/index.json
- **Neo4JImportPath** (*str*) – The path which Neo4J will use to import data
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **similarity** (*bool*) – Whether to use similarity as a measure to import or not
- **chebi_ids** (*list*) – A list of all the ChEBI_ID which are considered a reason to import
- **names** (*list*) – A list of all the Name which are considered a reason to import
- **hmdb_ids** (*list*) – A list of all the HMDB_ID which are considered a reason to import
- **inchis** (*list*) – A list of all the InChI which are considered a reason to import
- **mesh_ids** (*list*) – A list of all the MeSH_ID which are considered a reason to import

improve_search_terms(*driver*, *chebi_ids*, *names*, *hmdb_ids*, *inchis*, *mesh_ids*)

Improves the search terms already provided to the CanGraph programme by processing the text stings and finding synonyms in various platforms

Parameters

- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **chebi_ids** (*str*) – A string of ";" separated values of all the ChEBI_ID representing the current metabolite
- **names** (*list*) – A string of ";" separated values of all the Name representing the current metabolite
- **hmdb_ids** (*list*) – A string of ";" separated values of all the HMDB_ID representing the current metabolite
- **inchis** (*list*) – A string of ";" separated values of all the InChI representing the current metabolite
- **mesh_ids** (*list*) – A string of ";" separated values of all the MeSH_ID representing the current metabolite

Returns A list containing [chebi_ids, names, hmdb_ids, inchis, mesh_ids], with all their synonyms

Return type list

improve_search_terms_with_cts(*query*, *query_type*, *chebi_ids*, *names*, *hmdb_ids*, *inchis*, *mesh_ids*)

Improves the search terms already provided to the CanGraph programme by using The Chemical Translation Service to find synonyms in IDs

Parameters

- **query** (*str*) – The term we are currently querying for
- **query_type** (*str*) – The kind of query to search; one of ["ChEBI_ID", "HMDB_ID", "Name", "InChI", "MeSH_ID"]
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **chebi_ids** (*str*) – A string of ";" separated values of all the ChEBI_ID representing the current metabolite
- **names** (*list*) – A string of ";" separated values of all the Name representing the current metabolite
- **hmdb_ids** (*list*) – A string of ";" separated values of all the HMDB_ID representing the current metabolite
- **inchis** (*list*) – A string of ";" separated values of all the InChI representing the current metabolite
- **mesh_ids** (*list*) – A string of ";" separated values of all the MeSH_ID representing the current metabolite

Returns A list containing [chebi_ids, names, hmdb_ids, inchis, mesh_ids], with all their synonyms

Return type list

improve_search_terms_with_metanetx(*query*, *query_type*, *driver*, *chebi_ids*, *names*, *hmdb_ids*, *inchis*, *mesh_ids*)

Improves the search terms already provided to the CanGraph programme by using the MetaNetX web service to find synonyms in IDs

Parameters

- **query** (*str*) – The term we are currently querying for
- **query_type** (*str*) – The kind of query to search; one of ["ChEBI_ID", "HMDB_ID", "Name", "InChI", "MeSH_ID"]

- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **chebi_ids** (*str*) – A string of ";" separated values of all the ChEBI_ID representing the current metabolite
- **names** (*list*) – A string of ";" separated values of all the Name representing the current metabolite
- **hmdb_ids** (*list*) – A string of ";" separated values of all the HMDB_ID representing the current metabolite
- **inchis** (*list*) – A string of ";" separated values of all the InChI representing the current metabolite
- **mesh_ids** (*list*) – A string of ";" separated values of all the MeSH_ID representing the current metabolite

Returns A list containing [chebi_ids, names, hmdb_ids, inchis, mesh_ids], with all their synonyms

Return type list

`link_to_original_data`(*item_type*, *item*, *import_based_on*)

Links a recently-imported metabolite to the original data (that which caused it to be imported) by creating an `ÒriginalMetabolite` node that is `(n)-[r:ORIGINALLY_IDENTIFIED_AS]->(a)` related to the imported data

Parameters

- **tx** (*neo4j.Session*) – The session under which the driver is running
- **item_type** (*str*) – The property to match in the Neo4J DataBase
- **item** (*dict*) – The value of property `` `item_type` ``
- **import_based_on** (*list*) – A list of the methods that turned out to be valid for import, such as Name, ChEBI_ID...

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

`main`()

The function that executes the code

Note: This function disables rdkit's log messages, since rdkit seems to dislike the way some of the InChI strings it is getting from the databases are formatted

---

Todo: CAMBIAR NOMBRE A LOS MESH PARA INDICAR EL TIPO. AÑADIR NAME A LOS WIKIDATA

Todo: FIX THE REPEAT TRANSACTION FUNCTION

Todo: Match partial InChI based on DICE-MACCS

Todo: QUE FUNCIONE -> ACTUALMENTE ESTA SECCION RALENTIZA MAZO

Todo: CHECK APOC IS INSTALLED

Todo: FIX MAIN

Todo: MERGE BY INCHI, METANETX ID

Todo: Fix find_protein_interactions_in_metanetx

Todo: Mover esa funcion de setup a misc

Todo: EDIT conf.py

Todo: Document the following Schema Changes: * For Subject, we have a composite PK: Exposome_Explorer_ID, Age, Gender e Information * Now, more diseases will have a WikiData_ID and a related MeSH. This will help with networking. And, this diseases dont even need to be a part of a cancer! * The Gene nodes no longer exist in the full db? -> They do

3.8 CanGraph.miscelaneous module

A python module that provides a collection of functions to be used across the different scripts present in the CanGraph package, with various, useful functionalities

`call_db_schema_visualization()`

Shows the DB Schema. This function is intended to be run only in Neo4J's console, since it produces no output when called from the driver.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Todo: Make it download the image

`check_file(filepath)`

Checks for the presence of a file or folder. If it exists, it returns the filepath; if it doesn't, it raises an `argparse.ArgumentTypeError`, which tells argparse how to process file exclussion

Note: Perhaps its not ideal, but I will be using this also to check for file existence throughout the CanGraph project, although the error type might not be correct

Parameters **filepath** (*str*) – The path of the file or folder whose existence is being checked

Returns The original filepath, which now is sure to exist

Return type str

Raises **argparse.ArgumentTypeError** – If the file does not exist

check_neo4j_protocol(*string*)

Checks that a given `string` starts with any of the protocols accepted by the `neo4j.Driver`

Parameters **string** (*str*) – A string, which will normally represent the neo4j adress

Returns The same string that was provided as an argument (required by `argparse.ArgumentParser`)

Return type str

Raises **argparse.ArgumentTypeError** – If the string is not of the correct protocol

clean_database()

A CYPHER query that gets all the nodes in a Neo4J database and removes them, in transactions of 100 rows to alleviate memory load

Returns A text chain that represents the CYPHER query with the desired output. This can be run using: `neo4j.Session.run`

Return type str

Note: This is an **autocommit transaction**. This means that, in order to not keep data in memory (and make running it with a huge amount of data) more efficient, you will need to add ``:auto `` when calling it from the Neo4J browser, or call it as using `neo4j.Session.run` from the driver.

---

connect_to_neo4j(*port='bolt://localhost:7687'*, *username='neo4j'*, *password='neo4j'*)

A function that establishes a connection to the neo4j server and returns a `Driver` into which transactions can be passed

Parameters

- **port** (*str*) – The URL where the database is available to be queried. It must be of `bolt://` format
- **username** (*str*) – the username for your neo4j database; by default, `neo4j`
- **password** (*str*) – the password for your database; by default, `neo4j`

Returns An instance of Neo4J's Bolt Driver that can be used

Return type neo4j.Driver

Note: Since this is a really short function, this doesn't really simplify the code that much, but it makes it much more re-usable and understandable

---

countlines(*start*, *header=True*, *lines=0*, *begin_start=None*)

A function that counts all the lines of code present in a given directory; useful to show off in Sphinx Docs

Parameters

- **start** (*str*) – The directory from which to start the line counting

- **header** (*bool*) – whether to print a header, or not
- **lines** (*int*) – Number of lines already counted; do not fill, only for recursion
- **begin_start** (*str*) – The subdirectory currently in use; do not fill, only for recursion

Returns The number of lines present in `start`

Return type int

See also:

This function was taken from StackOverflow #38543709

`create_n10s_graphconfig()`

A CYPHER query that creates a *neosemantics* (n10s) constraint to hold all the RDF we will import.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

See also:

More information on this approach can be found in Neosemantics' 101 Guide and in Neo4J's guide on how to import data from Wikidata , where this approach was taken from

Deprecated since version 0.9: Since we are importing based on apoc.load.jsonParams, this is not needed anymore

`download(url, folder)`

Downloads a file from the internet into a given folder

Parameters

- **url** (*str*) – The Uniform Resource Locator for the Zipfile to be downloaded and unzipped
- **folder** (*str*) – The folder under which the file will be stored.

Returns The path where the file we just downloaded has been stored

Return type str

`download_and_untargz(url, folder)`

Downloads and unzips a given `tar.gz` from the internet

Parameters

- **url** (*str*) – The Uniform Resource Locator for the `tar.gz` to be downloaded and unzipped
- **folder** (*str*) – The folder under which the file will be stored.

Returns This function downloads and unzips the file in the desired folder, but does not produce any particular return.

`download_and_unzip(url, folder)`

Downloads and unzips a given Zipfile from the internet; useful for databases which provide zip access.

Parameters

- **url** (*str*) – The Uniform Resource Locator for the Zipfile to be downloaded and unzipped
- **folder** (*str*) – The folder under which the file will be stored.

Returns This function downloads and unzips the file in the desired folder, but does not produce any particular return.

See also:

Code snippets for this function were taken from Shyamal Vaderia's Github and from StackOverflow #32123394

export_graphml(*exportname*)

Exports a Neo4J graph to GraphML format. The graph will be exported to Neo4JImportPath

Parameters **exportname** (*str*) – The name for the exported file, which will be saved under ./Neo4JImportPath/

Returns

A Neo4J connexion to the database that exports the file, using batch optimizations and smaller batch sizes to try to keep the impact on memory use low

Return type neo4j.Result

Note: for this to work, you HAVE TO have APOC availaible on your Neo4J installation

---

get_import_path(*driver*)

A function that runs an autocommit transaction to get Neo4J's Import Path

---

Note: By doing the Neo4JImportPath search this way (in two functions), we are able to run the query as a :obj: execute_read, which, unlike autocommit transactions, allows the query to be better controlled, and repeated in case it fails.

Parameters **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use

Returns Neo4J's Import Path, i.e., where Neo4J will pick up files to be imported using the `file:// ` schema

Return type str

import_graphml(*importname*)

Imports a GraphML file into a Neo4J graph. The file has to be located in Neo4JImportPath

Parameters **importname** (*str*) – The name for the file to be imported, which must be under ./Neo4JImportPath/

Returns

A Neo4J connexion to the database that imports the file, using batch optimizations and smaller batch sizes to try to keep the impact on memory use low

Return type neo4j.Result

Note: for this to work, you HAVE TO have APOC availaible on your Neo4J installation

---

kill_neo4j(*neo4j_home='neo4j'*)

A simple function that kills any process that was started using a cmd argument including "neo4j"

Parameters **neo4j_home** (*str*) – the installation directory for the `neo4j` program; by default, `neo4j`

Warning: This function may unintendedly kill any command run from the `neo4j` folder. This is unfortunate, but the creation of this function was essential given that `neo4j stop` does not work properly; instead of dying, the process lingers on, interfering with *find_neo4j_installation_status* and hindering the main program

manage_transaction(*tx*, *driver*, *num_retries=10*, *neo4j_home='neo4j'*, ***kwargs*)

A function that repeats transactions whenever an error is found. This may make an incorrect script unnecessarily repeat; however, since the error is printed, one can discriminate those out, and the function remains helpful to prevent SPARQL Read Time-Outs.

It will also re-start neo4j in case it randomly dies while executing a query.

Parameters

- **tx** (*str*) – The transaction that we desire to run, specified as a CYPHER query
- **driver** (*neo4j.Driver*) – Neo4J's Bolt Driver currently in use
- **num_retries** (*int*) – The number of times that we wish the transaction to be retried
- **neo4j_home** (*str*) – the installation directory for the `neo4j` program; by default, `neo4j`
- ****kwargs** – Any number of arbitrary keyword arguments

Raises **Exception** – An exception telling the user that the maximum number of retries has been exceded, if such a thing happens

Returns The response from the Neo4J Database

Return type list

Note: This function does not accept args, but only kwargs (named keyword arguments). Thus, if you wish to add a parameter (say, `number`, you should add it as: `number=33`

---

merge_duplicate_nodes(*node_types*, *node_property*, *optional_condition=''*, *more_props=''*)

Removes any two nodes of any given `` `node_type` `` with the same `` `condition` ``.

Parameters

- **node_types** (*str*) – The labels of the nodes that will be selected for merging; i.e. `n:Fruit OR n:Vegetable`
- **node_property** (*str*) – The node properties used for collecting, if not using all properties.
- **optional_condition** (*str*) – An optional Neo4J Statement, starting with "AND", to be added after the `WHERE` clause.

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Warning: When using, take good care on how the keys names are written: sometimes, if a key is not present, all nodes will be merged!

old_sleep_with_counter(*seconds*, *step=20*, *message='Waiting...'*)

A function that waits while showing a cute animation, but **without using the ``alive_progress` module**

Note: This function interacts weirdly with slurn; I'd recommend to not use it on the HPC

Parameters

- **seconds** (*int*) – The number of seconds that we would like the program to wait for
- **step** (*int*) – The number times the counter wheel will turn in a second; by default, 20
- **message** (*str*) – An optional, text message to add to the waiting period

purge_database(*driver*, *method=['merge', 'delete']*)

A series of commands that purge a database, removing unnecessary, duplicated or empty nodes and merging those without required properties. This has been converted into a common function to standarize the ways the nodes are merged.

Args: driver (neo4j.Driver): Neo4J's Bolt Driver currently in use method (list): The part of the function that we want to execute; if ["delete"], only call

queries that delete nodes; if ["merge"], only call those that merge; if both, do both

Returns This function modifies the Neo4J Database as desired, but does not produce any particular return.

Warning: When modifying, take good care on how the keys names are written: with *merge_duplicate_nodes*, sometimes, if a key is not present, all nodes will be merged!

remove_ExternalEquivalent()

Removes all nodes of type: ExternalEquivalent from he DataBase; since this do not add new info, one might consider them not useful.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

remove_duplicate_relationships()

Removes duplicated relationships between ANY existing pair of nodes.

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Note: Only deletes DIRECTED relationships between THE SAME nodes, combining their properties

See also:

This way of working has been taken from StackOverflow #18724939

remove_n10s_graphconfig()

Removes the "_GraphConfig" node, which is necessary for querying SPARQL endpoints but not at all useful in our final export

Parameters **tx** (*neo4j.Session*) – The session under which the driver is running

Returns A Neo4J connexion to the database that modifies it according to the CYPHER statement contained in the function.

Return type neo4j.Result

Deprecated since version 0.9: Since we are importing based on apoc.load.jsonParams, this is not needed anymore

restart_neo4j(*neo4j_home='neo4j'*)

A simple function that (re)starts a neo4j server and returns its bolt adress

Parameters **neo4j_home** (*str*) – the installation directory for the `neo4j` program; by default, `neo4j`

Note: Re-starting is better than starting, as it tries to kills old sessions (a task at which it fails miserably, thus the need for *kill_neo4j*), and, most importantly, because it returns the currently used bolt port

---

scan_folder(*folder_path*)

Scans a folder and finds all the files present in it

Parameters **folder_path** (*str*) – The folder that is to be scanned

Returns A list of all the files in the folder, listed by their absolute path

Return type list

sleep_with_counter(*seconds*, *step=20*, *message='Waiting...'*)

A function that waits while showing a cute animation

Parameters

- **seconds** (*int*) – The number of seconds that we would like the program to wait for
- **step** (*int*) – The number times the counter wheel will turn in a second; by default, 20
- **message** (*str*) – An optional, text message to add to the waiting period

split_csv(*filename*, *folder*, *sep=','*, *sep_out=','*, *startFrom=0*, *withStepsOf=1*)

Splits a given .csv/tsv file in n smaller csv files, one for each row on the original file, so that it does not crash when processing it. It also allows to start reading from `startFrom` lines

Parameters

- **filepath** (*str*) – The path to the file that needs to be xplitted
- **splittag** (*str*) – The tag based on which the file will be split
- **bigtag** (*str*) – The main tag of the file, which needs to be re-added.

Returns The number of files that have been produced from the original

Return type int

Warning: The original file will be removed

split_xml(*filepath*, *splittag*, *bigtag*)

Splits a given .xml file in n smaller XML files, one for each `splittag` section that is pressent in the original file, which should be of type `bigtag`. For example, we might have an <hmdb> file which we want to slit based on the <metabolite> items therein contained. Ths is so that Neo4J does not crash when processing it.

Parameters

- **filepath** (*str*) – The path to the file that needs to be xplitted
- **splittag** (*str*) – The tag based on which the file will be split
- **bigtag** (*str*) – The main tag of the file, which needs to be re-added.

Returns The number of files that have been produced from the original

Return type int

Warning: The original file will be removed

untargz(*file_path*, *folder*)

Untargzs a file present at a given `file_path` into a given `folder`

Parameters

- **url** (*str*) – The Uniform Resource Locator for the Tarfile to be untargz
- **folder** (*str*) – The folder under which the file will be stored.

Returns The path where the file we just untargz has been stored

Return type str

unzip(*file_path*, *folder*)

Unizps a file present at a given `file_path` into a given `folder`

Parameters

- **url** (*str*) – The Uniform Resource Locator for the Zipfile to be unzipped
- **folder** (*str*) – The folder under which the file will be stored.

Returns The path where the file we just unzipped has been stored

Return type str

3.9 CanGraph.setup module

A python module that prepares the local environment, to be able to run the *main* and *deploy* functions. This can be either run in an interactive way, requiring user input; or in a automatic way, in order to pre-configure things, for example, if you are using the singularity package

3.9.1 CanGraph.setup Usage

To use this module:

A python module that prepares the local environment, to be able to run the CanGraph.main and CanGraph.deploy functions.

```
usage: python3 setup.py [-h] [-i] [-a] [--dbfolder [DBFOLDER]] [--git [GIT]]
                        [--requirements [REQUIREMENTS]] [-n [NEO4J]]
                        [--neo4j_username [NEO4J_USERNAME]]
                        [--neo4j_password [NEO4J_PASSWORD]]
```

Named Arguments

-i, --interactive	tells the script if it wants interaction from the user and information shown to them; similar to –verbose
-a, --all	runs all the options below at once; equivalent to -dgnr; it DOES NOT activate the interactive mode
--dbfolder	set up the databases from which the program will pull its info using the provided folder
--git	prepare the git environment for the deploy script using the provided git folder
--requirements	installs all the requirements needed for all the possible options from the given requirements file
-n, --neo4j	set up the neo4j local environment, to run from the provided folder
--neo4j_username	the username for the neo4j database
--neo4j_password	the password for the neo4j database

3.9.2 CanGraph.setup Functions

This module is comprised of:

`args_parser()`

Parses the command line arguments into a more usable form, providing help and more

Returns A dictionary of the different possible options for the program as keys, specifying their set value. If no command-line arguments are provided, the help message is shown and the program exits.

Return type argparse.ArgumentParser

Note: Note that, in Google Docstrings, if you want a multi-line `Returns` comment, you have to start it in a different line :(

Note: The return **must** be of type `argparse.ArgumentParser` for the `argparse` directive to work and auto-gen docs

---

Note: The `--all```option has to be adressed outside of this function in order to not mess up the ``argparse` directive in sphinx

Note: By using `argparse.const` instead of `argparse.default`, the check_file function will check "" (the current dir, always exists) if the arg is not provided, not breaking the function; if it is, it checks it.

change_neo4j_password(*new_password*, *old_password='neo4j'*, *user='neo4j'*, *database='system'*, *neo4j_home='neo4j'*)

Changes the neo4j password for user `user`, from `old_password` to `new_password`, by using a simple query in cypher-shell

Parameters

- **neo4j_home** (*str*) – the installation directory for the `neo4j` program; by default, `neo4j`
- **new_password** (*str*) – the new password for the database
- **old_password** (*str*) – the old password for the database, needed for identification.
- **user** (*str*) – the user for which the password is being changed.
- **database** (*str*) – the name of the database for which we want to modify the password. By default, it is `system`, since Neo4J's community edition only allows for one database

Warning: DO NOT REMOVE THE TRY-EXCEPT BLOCK THAT ATTEMPTS TO CONNECT TO NEO4J: It somehow magically maked the passsword change work. IT WILL NOT WORK IF THAT LINES ARE NOT PRESENT

check_exposome_files(*databasefolder='./DataBases'*)

Checks for the presence of all the files that should be in "`databasefolder`/ExposomeExplorer"` for the ExposomeExplorer part of the script to run

Parameters **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found

Returns One of ["Splitted, "UnSplitted", "Error"]. If "Error", Exposome-Explorer should not be used as a data source; if "UnSplitted", please split the "components" file.

Return type str

configure_neo4j(*neo4j_home='neo4j'*)

Modifies the Neo4J conf file according to some recommendations provided by `memrec`, neo4j's memory recommendator. It also enables the Awesome Procedures On Cypher (APOC) plugin from Neo4j Labs, and enables other basic confs such as file export and import or bigger timeouts

Parameters **neo4j_home** (*str*) – the installation directory for the `neo4j` program; by default, `neo4j`

Note: In order to make the setup more consistent, this function also forces the Neo4JImportPath (`dbms.directories.import`) to be presented in an absolute way, instead of being relative to `neo4j_home`

final_message(*interactive=False*)

Prompts the user with a final message.

Parameters **interactive** (*bool*) – Whether the session is set to be interactive or not

find_neo4j_installation_status(*neo4j_home='neo4j'*, *neo4j_username='neo4j'*, *neo4j_password='neo4j'*)

Finds the installation status of Neo4J by trying to use it normally, and analyzing any thrown exceptions

Parameters

- **neo4j_home** (*str*) – the installation directory for the neo4j program; by default, neo4j
- **neo4j_username** (*str*) – the username for the neo4j database; by default neo4j
- **neo4j_password** (*str*) – the password for the neo4j database; by default neo4j

Returns A list of two booleans: whether neo4j exists at neo4j_home, and whether the supplied credentials are valid or not

Return type list

initial_message()

Prompts the user with an initial message if the session is set to be interactive.

Parameters **interactive** (*bool*) – Whether the session is set to be interactive or not

install_neo4j(*neo4j_home='neo4j'*, *interactive=False*, *version='4.4.0'*)

Installs the neo4j database program in the neo4j_home folder, by getting it from the internet according to the Operating System the script is been run in (aims for multi-platform!)

Parameters

- **neo4j_home** (*str*) – the installation directory for the neo4j program; by default, neo4j
- **interactive** (*str*) – tells the script if it wants interaction from the user and information shown to them
- **version** (*str*) – the version of the neo4j software that we wish to install

install_packages(*requirements_file=None*, *package_name=None*, *interactive=False*)

Automates installing packages using PIP

Parameters

- **requirements_file** (*str*) – The path to a "requirements.txt" file, containing one requirement per line
- **package_name** (*str*) – A package to be installed
- **interactive** (*bool*) – Whether the session is set to be interactive or not

Raises ValueError – If neither a requirements_file nor a package_name is provided

main()

The function that executes the code

setup_database_index(*databasefolder='./DataBases'*)

Prepares the index file for all the databases present in the databasefolder folder, which will helpfully reduce processing time a lot

Parameters **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found

Returns A dictionary containing the index for all the databases in databasefolder. This index will be written as JSON in databasefolder/index.json

Return type dict

setup_databases(*databasefolder='./DataBases'*, *interactive=False*)

Set Up the databasefolder from where the *main* script will take its data. It does so by creating or removing and re-creating the databasefolder, and putting inside it, or asking/checking if the user has put inside, the necessary files

Parameters

- **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found
- **interactive** (*bool*) – Whether the session is set to be interactive or not

setup_drugbank(*databasefolder='./DataBases'*, *interactive=False*)

Sets up the files relative to the SMPDB database in the databasefolder, splitting them for easier processing later on.

Parameters

- **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found
- **interactive** (*bool*) – Whether the session is set to be interactive or not

Returns True if everything went okay; False otherwise. If False, DrugBank should not be used as a data source

Return type bool

Warning: When updating the DrugBank DataBase Version, please edit this function to reflect the correct number of files

setup_exposome(*databasefolder='./DataBases'*, *interactive=False*)

Sets up the files relative to the Exposome Explorer database in the databasefolder, splitting them for easier processing later on. If the session is set to be interactive, the user will be given time to add the files themselves; if not, the full suite of necessary files will be checked for their presence in databasefolder

Then, the "components" file will be splitted into one record oer line, as *main* requires

Parameters

- **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found
- **interactive** (*bool*) – Whether the session is set to be interactive or not

Returns True if everything went okay; False otherwise. If False, Exposome-Explorer should not be used as a data source

Return type bool

Warning: When updating the Exposome Explorer DataBase Version, please edit *check_exposome_files* to reflect the correct number of files

setup_folders(*databasefolder='./DataBases'*, *interactive=False*)

Creates the databasefolder if it does not exist. If it does, it either asks before overwriting in interactive mode, or directly overwrites in auto mode.

Parameters

- **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found
- **interactive** (*bool*) – Whether the session is set to be interactive or not

Raises ValueError – If the Databases folder already exists (so as not to overwrite)

Returns True if successful, False otherwise.

Return type bool

setup_git(*path_to_repo='.git'*)

Set Up the Git environment for the *deploy* script. It does so by removing any existing remotes and setting two new ones: github and codeberg, with their respective branches

Parameters **path_to_repo** (*str*) – The path to the Git repo; by default, `.git`

setup_hmdb(*databasefolder='./DataBases'*)

Sets up the files relative to the HMDB database in the `databasefolder`, splitting them for easier processing later on.

Parameters **databasefolder** (*str*) – The main folder where all the databases we will be using are to be found

Returns True if everything went okay; False otherwise. If False, DrugBank should not be used as a data source

Return type bool

> **Warning:** When updating the Exposome Explorer DataBase Version, please edit *check_exposome_files* to reflect the correct number of files

setup_neo4j(*neo4j_home='neo4j'*, *neo4j_username='neo4j'*, *neo4j_password='neo4j'*, *interactive=False*)

Sets ups the neo4j environment in `neo4j_home`, so that the functions in *main* can propperly function. Using the functions present in this module, it finds if neo4j is installed with default credentials, and, if not, it installs it, changing the default password to a new one, and returning its value

Parameters

- **neo4j_home** (*str*) – the installation directory for the `neo4j` program; by default, `neo4j`
- **neo4j_username** (*str*) – the username for the neo4j database; by default `neo4j`
- **neo4j_password** (*str*) – the password for the neo4j database; by default `neo4j`
- **interactive** (*str*) – tells the script if it wants interaction from the user and information shown to them

Returns The password that was set up for the new neo4j database. This is also written to .neo4jpassword

Return type str

Note: This has been designed to be used with a `neo4j_home` located in the WorkDir, but can be used in any other location with read/write access, or even with `apt install` installed versions! Just find its `neo4j_home`, make sure it has r/w access, and provide it to the program!

Note: If no neo4j_password is provided or if neo4j_password = "neo4j", the function will check for a previously created ".neo4jpassword" file, signalling a possible pre-existing database with known credentials

`setup_smpdb`(*databasefolder='./DataBases'*)

Sets up the files relative to the SMPDB database in the `databasefolder`, splitting them for easier processing later on.

Parameters **`databasefolder`** (*str*) – The main folder where all the databases we will be using are to be found

Returns True if everything went okay; False otherwise. If False, DrugBank should not be used as a data source

Return type bool

`update_neo4j_confs`(*key*, *value*, *conf_file='neo4j/conf/neo4j.conf'*)

Updates a preference on neo4j's `conf_file`, given its name (`key`) and its expected `value` If a preference is set with a value other than `key`, said value will be overwritten; if it is commented, it will be uncommented (thanks to regex!)

Parameters

- **`conf_file`** (*str*) – The location for ne4j's configuration file; usually, it should be `neo4j/conf/neo4j.conf`
- **`key`** (*str*) – The key for neo4j's configuration parameter that is being set up
- **`value`** (*str*) – The value for said parameter

CHAPTER

FOUR

TO-DO LIST

The following problems are known issues which will be solved, but are yet to be adressed:

Todo: When fixing queries, fix the main subscript also

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.annotate_using_wikidata, line 8.)

Todo: CAMBIAR NOMBRE A LOS MESH PARA INDICAR EL TIPO. AÑADIR NAME A LOS WIKIDATA

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 6.)

Todo: FIX THE REPEAT TRANSACTION FUNCTION

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 7.)

Todo: Match partial InChI based on DICE-MACCS

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 8.)

Todo: QUE FUNCIONE -> ACTUALMENTE ESTA SECCION RALENTIZA MAZO

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 9.)

Todo: CHECK APOC IS INSTALLED

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 10.)

Todo: FIX MAIN

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 11.)

Todo: MERGE BY INCHI, METANETX ID

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 12.)

Todo: Fix find_protein_interactions_in_metanetx

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 13.)

Todo: Mover esa funcion de setup a misc

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 14.)

Todo: EDIT conf.py

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 15.)

Todo: Document the following Schema Changes: * For Subject, we have a composite PK: Exposome_Explorer_ID, Age, Gender e Information * Now, more diseases will have a WikiData_ID and a related MeSH. This will help with networking. And, this diseases dont even need to be a part of a cancer! * The Gene nodes no longer exist in the full db? -> They do

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/main.py:docstring of CanGraph.main.main, line 17.)

Todo: Make it download the image

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/miscelaneous.py:docstring of CanGraph.miscelaneous.call_db_schema_visualization, line 7.)

Todo: In some other parts of the script, sequences are being added as properties on Protein nodes. A common format should be set.

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifyDrugBank/build_datab of CanGraph.GraphifyDrugBank.build_database.add_sequences, line 12.)

Todo: Investigate https://stackoverflow.com/questions/14026217/using-neo4j-distinct-and-order-by-on-different-properties

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifyDrugBank/build_datab of CanGraph.GraphifyDrugBank.build_database.add_targets_enzymes_carriers_and_transporters, line 26.)

Todo: It would be nice to be able to distinguish between experimental and predicted properties

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifyHMDB/build_database
of CanGraph.GraphifyHMDB.build_database.add_biological_properties, line 16.)

Todo: It would be nice to be able to distinguish between experimental and predicted properties

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifyHMDB/build_database
of CanGraph.GraphifyHMDB.build_database.add_experimental_properties, line 16.)

Todo: It would be nice to be able to distinguish between experimental and predicted properties

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifyHMDB/build_database
of CanGraph.GraphifyHMDB.build_database.add_predicted_properties, line 16.)

Todo: This file is really big. It could be divided into smaller ones.

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifySMPDB/build_databas
of CanGraph.GraphifySMPDB.build_database.add_pathways, line 12.)

Todo: Why is the SMPDB_ID property called like that and not SMDB_ID?

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/GraphifySMPDB/build_databas
of CanGraph.GraphifySMPDB.build_database.add_proteins, line 15.)

Todo: Might include P684 “Orthologues” for more info (it crashed java)

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/QueryWikidata/build_database.
of CanGraph.QueryWikidata.build_database.add_gene_info, line 13.)

Todo: Might include P527 “has part or parts” for more info (it crashed java)

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/QueryWikidata/build_database.
of CanGraph.QueryWikidata.build_database.add_metabolite_info, line 17.)

Todo: ADD ROLE to metabolite interactions

(The original entry is located in /home/pablo/Documentos/Trabajos del cole/UNI/IARC/Work/CanGraph/QueryWikidata/build_database.
of CanGraph.QueryWikidata.build_database.add_more_drug_info, line 13.)

A utility to study and analyse cancer-associated metabolites using knowledge graphs

Get started

CanGraph is a python program that allows you to extract information about a newly discovered or existing metabolite from the following databases:

- Human Metabolome DB
- DrugBank
- Exposome Explorer
- WikiData
- SMPDB
- MesH and MetaNetX

For this purpose, CanGraph accepts any of the following inputs:

- **InChI:** The International Chemical Identifier for the metabolite, which can be calculated using Open Source Software and identifies a metabolite with 99.99% accuracy based on its structure
- **InChIKey:** The Hashed, shortened version of the InChI, sometimes used for efficiency
- **Name:** A commonly accepted name for the metabolite; preferably, IUPAC's standarized name
- **HMDB_ID:** The Metabolite's Identifier in the Human Metabolome Database
- **ChEBI_ID:** The Metabolite's Identifier in the ChEBI Database

which must be provided as explained in *CanGraph's README*

CHAPTER

FIVE

ACKNOWLEDGEMENTS

This research was funded by the Agence Nationale de la Recherche Project number ANR-19-CE45-0021 (New approaches to bridge the gap between genome-scale metabolic networks and untargeted metabolomics – **MetClassNet**) and the Deutsche ForschungsGemeinschaft Project number 431572533 (**MetClassNet**: new approaches to bridge the gap between genome-scale metabolic networks and untargeted metabolomics)

You can check MetClassNet's Official Website here

Most of the work for this project has been carried out at the International Agency for Research on Cancer's OncoMetaBolomics Team

PYTHON MODULE INDEX

C

INDEX

A

B

C

D

E

F

G

I

K

L

M

O

P

R

S

U

W