LCA analysis of metagenomic sequences in Python

Development Journal

Tobias Lass

Contents

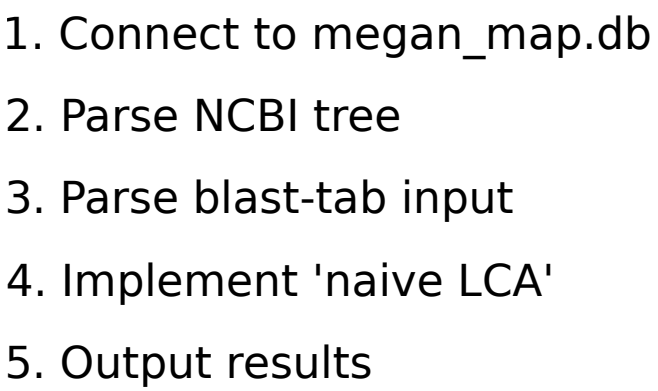

Figure 1: General idea of the process of the practical part.

3 Planning

3.1 Project Pipeline

2021-03-09

In the first meeting with Prof. Dr. Huson and PhD student Timo Lucas on March 8 we discussed the content of the thesis, glanced over some technical details and worked out an order of tasks to accomplish. I later received the meeting notes and the transcript of a *sqlite* demo. The general idea of the practical part is to perform an LCA analysis of a set of metagenomic sequences, which are for the purpose of this thesis contained within a *blast-tab* file.

As illustrated in fig. 1, the project is structured into five steps. For the first step I have to connect my program to the latest *megan_map.db*, which contains a mapping of *accessions* to *taxonomy ids*. For that purpose I was recommended to use sqlite. Next I am to parse the NCBI taxonomy tree into a useable format, which maps the taxonomy ids to nodes in an internal tree. In step three I have to extract the accession and score of sequences from the user provided blast-tab file. With the accessions from step three, accession to taxonomy id from step one and full taxonomy tree from step two, I can perform a LCA analysis in step four. Finally, the output of the analysis is returned and maybe visualised in step five.

For the written thesis itself I am to provide a proof of concept of my program by performing a LCA analysis on a blast-tab dataset and then compare it to its Java implementation by performance, usability, result quality, etc.

3.2 Program Architecture

2021-03-10

In order to reduce rewriting a ton of code I will try to thoroughly plan my code ahead. First, a general pipeline from input to output in fig. 2. I tried to create a flow diagram of where data goes and to group by tasks. Not very formal nor clean, but some observations can be made.

Observation 1: the megan map database is several gigabyte big, so I should not load it as dictionary in its entirety. I also need to access the database only once per blast input, so I do not have to maintain an open connection.

Observation 2: the parsing of NCBI taxonomy data to a phylogenetic tree is independent from the user input. Therefore I probably should do it only once per program use. Or even better, I should parse the data the very first time I use the program and use my own internal representation for every subsequent usage. The user would only need either the NCBI data or my internal representation. If the NCBI data is updated, delete my internal representation and parse it again.

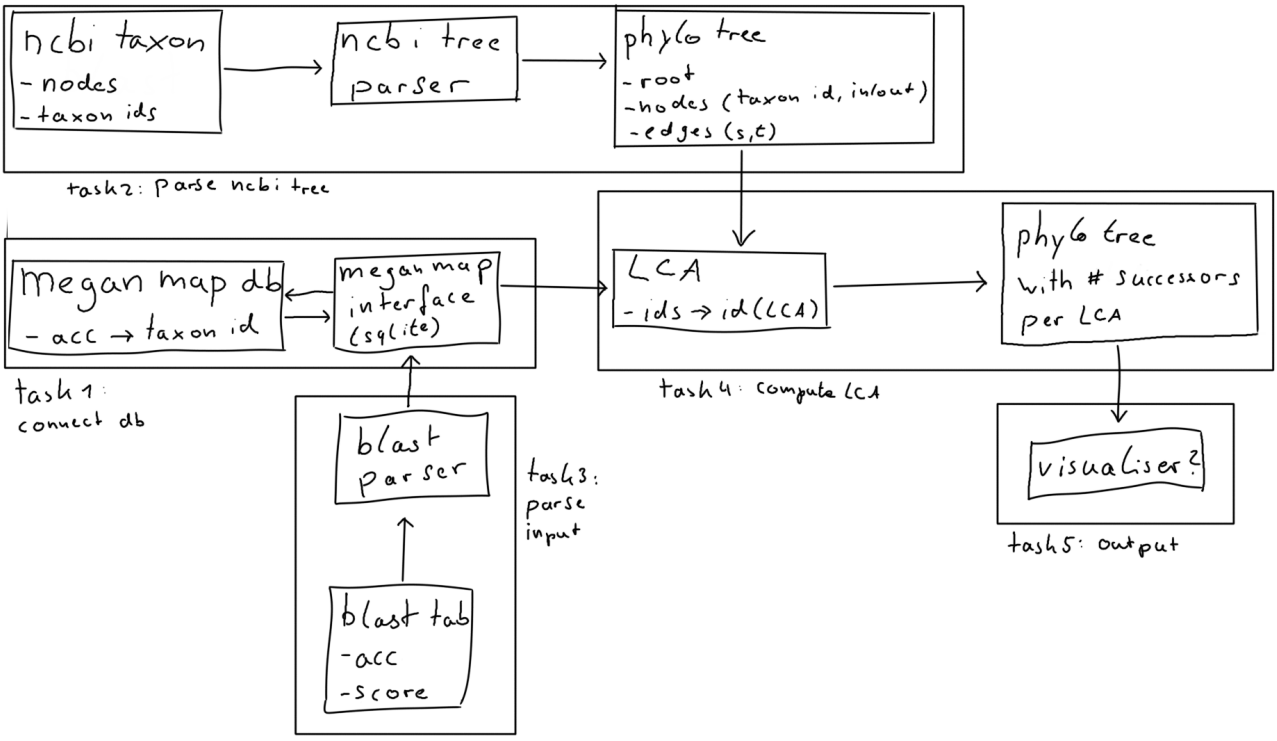

Figure 2: First draft of program structure.

Observation 3: the LCA algorithm takes an 'empty' phylogenetic tree and fills it with information computed from the blast input. It shall also be observed that now that I am thinking about it, I am not entirely sure exactly how or with what information the phylogenetic tree is to be filled, but those are questions for task 4 and 5. From the meeting I gathered that the information mapped to nodes in the tree should be how many reads inherit from a node, which seemed clear in the moment. I will investigate this at a later point.

For what ever the runner or main function should be that stitches all blocks together, I am not sure yet. For the purpose of this thesis it is going to be a command line application, but how ever I design it, it will be discarded immediately once someone new works on my code to expand it. It would be integrated into a system that provides a GUI and command line usability, so the best I can do for my successor is to make my code as pluggable as possible. It would also spare me the excruciating pain of writing a sophisticated arg parser let alone design a GUI.

The modularity of my code and the possibility of splitting it up into entirely separate packages was also discussed in the meeting. I think Java MEGAN using a 'third party' data structure makes sense, so maybe making a package for parsing the NCBI taxonomy tree to an internal phylogenetic tree and the functionality to save/load it would make sense too. Maybe this could be expanded with a phylogenetic tree visualiser or what ever data it contains visualiser.

A blast parser for various blast formats could also be a stand-alone thing. Just a tiny module that parses blast files to a dictionary or something. I am not sure about the megan map database interface though. Its purpose is really specific, but I suppose it does connect well to the blast parser. An entire package may provide a functionality to parse blast files and retrieve a dictionary of taxonomy ids, their counts and scores or something.

At the core of everything stands the LCA algorithm. Let's say the program takes a blast file as input and uses the blast/megan map package to parse the input and receive the perfect input of the LCA algorithm. At an independent point in time the program has already loaded a blank phylogenetic tree from the NCBI/phylogenetic tree package and can now fill it with information retrieved from the LCA

algorithm. The output can then be plugged into the visualisation functions of the NCBI/phylogenetic tree package while LCA is computing the next blast input. Does this sound modular enough for proper code expansion? I see great opportunities for multi threading and offloading tasks to servers.

I think it would work great because with the input handling being outside of the core functionalities, a top level program could maintain a variety of algorithms besides LCA and gather/plug data wherever desired. A question though that came to mind is: shouldn't the computationally heavy tasks be performed in a more efficient programming language than Python? No doubt that stitching things together with Python will work out great, but maybe in the future there would be an API between the Python runner and algorithms in C++, Java or Rust or something. I believe this is beyond the scope of my thesis, but I shall discuss this with my supervisors at some point.

4 Task 1: Connect to megan_map.db

4.1 Framework

2021-03-10

What goes in, what comes out? In order to keep things as modular as possible, I need to reduce the 'surface of the API'. It shall receive only absolutely necessary input and let what ever runner is stitching everything together figure out the rest. That means, my megan map interface will receive a list of accessions as input and output a map of accessions to taxonomy ids. It may not be the most efficient implementation for the runner, but it would be fairly robust. This part receives input generated by the blast parser or what the runner takes from the blast parser. If it was a stand-alone package the top level runner would simply take a blast file and the location of the db as input, parse the blast file, map the accessions and be done with it. Seems good enough I think, it should be straight forward.

4.2 megan_map.db

2021-03-10

This module is only going to be an interface to the megan_map.db where the actual data is stored. In what form is it available? Well right now it's a 7.5 GB heavy file on my hard drive. Definitely too much to load into the program in its entirety. However I implement this, I can only send an inquiry via *sqlite3* or something to the database and not to an internal dictionary. That is okay though, because it appears to be really fast. I feel like having to download a 7.5 GB file to use the program is an inconvenience for the user, maybe the database could be hosted on a server? Never mind, I just googled it and sqlite is 'serverless'. I was told this in the meeting too and just kind of forgot. I think I actually found an accession to taxonomy id map on the NCBI website, but if I remember correctly from GBI, get requests to that place are kind of slow? Would that even be worth it? Anyway let's assume the file is on the disk of the user and it comes separately from the program. I think I have two options:

Option 1: hard code the relative path from module to database. That would make things really easy for me, but then again I would have to catch errors with a butterfly net and provide missing file troubleshooting. In that case I would only have to connect to the database via sqlite3 once when loading the module. I am not convinced connecting is expensive enough to warrant this approach.

Option 2: create an instance of db-interface given the path to the database. Handling missing file stuff may be done somewhere else in the runner then. I would only have to connect to the database once and not deal with the file. The con of this would be having to maintain the db-interface instance somewhere. Imagine this was a package: why would I want an instance of this thing? Just provide the functionality!

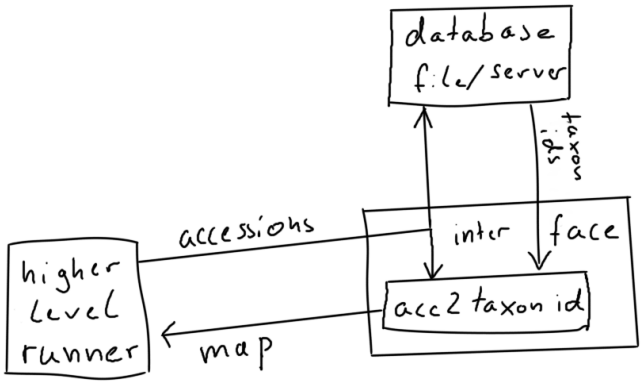

Figure 3: General idea of megan_map.db interface.

Option 3: similar to option 2, but static. Just a module with a bunch of functions. I would have to connect to the database for every inquiry, but I think it's okay. This seems to be the most modular and elegant solution. I would still have to fish for errors in a real world application, but I do not think it would be all that bad.

4.3 Implementation

2021-03-10

Option 3 it is I suppose. What remains unclear is: database file on the disk or potentially receive data from the NCBI server? Anyway I think I the process will be: receive accessions → open connection to db → retrieve corresponding taxonomy ids → create a dictionary → close connection → return dictionary as illustrated in fig. 3. Let's get more specific.

- **get_accessions2taxonids:** top level function. Receives a list of accessions and the location of the database. Calls **connect**, calls **map_accessions**, calls **disconnect**, then returns **accessions2taxonids**.
- **connect:** receives a database location and returns a sqlite3 connection to the database.
- **map_accessions:** receives a database cursor and creates a dictionary entry per accession via sqlite3 query. Returns the map.
- **disconnect:** disconnects from the database.

I just figured out that I can use `'select Taxonomy from mappings where Accession in <tuple>'` to fetch all my queries at once. Originally **inquire** was to fetch a single taxonomy id from an accession, but that is not necessary any more. It will return the full list instead. I just have to trust that the accessions list and taxonomy ids list maintain their order. According to stackoverflow lists and tuples do maintain orders while hashed collections like set and dictionary do not. Originally I wanted to be fancy and type the accessions input `Iterable[str]`, but I guess it has to be a list realistically.

Never mind, I just acquired new information again. I can use `'select Accession, Taxonomy from mappings where Accession in <tuple>'` to pretty much get the entire mapping immediately. That means I can keep accessions as iterable and should probably remove **inquire** all together, because the extra step would be pretty inefficient. Amazing stuff really. I actually learned quite a bit about SQL queries now. Maybe I should have looked into that before writing my code.