LCA analysis of metagenomic sequences in Python

Development Journal

Tobias Lass

Contents

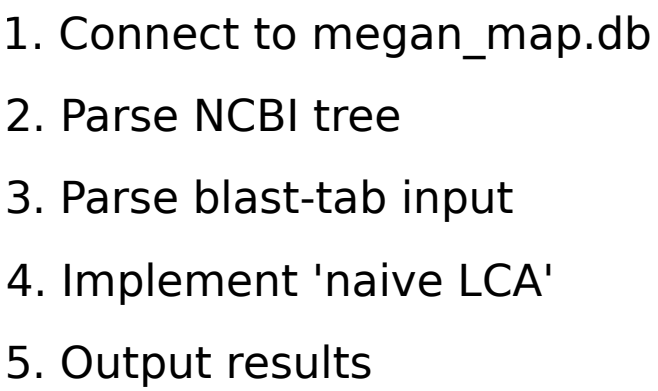

Figure 1: General idea of the process of the practical part.

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.

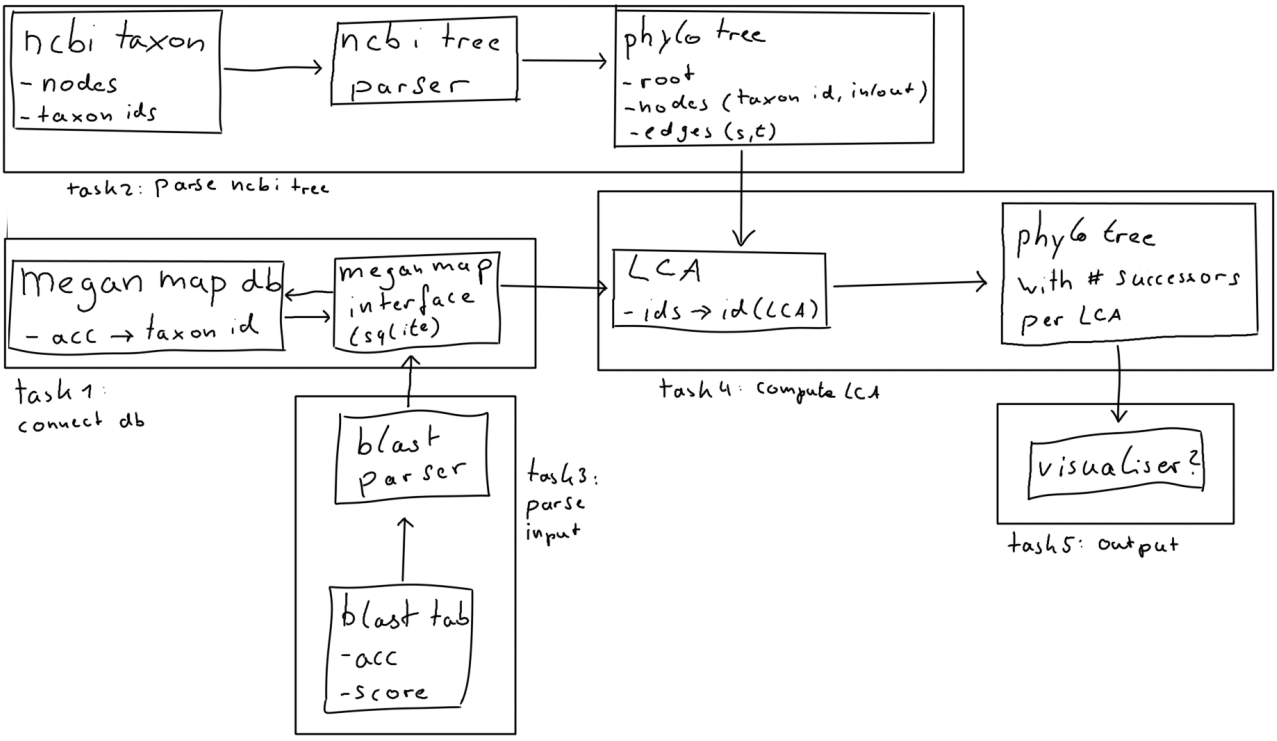

Figure 2: First draft of program structure.

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.

3.3 Feedback & Adjustments

2021-03-15

Feedback was positive. I found out that my taxonomy tree parsing should include GTDB and Silva too besides NCBI and that's the only adjustment to my plan. A second Bachelor student might use my blast → accession2taxonomy functionality in the future, so I'll keep that in mind.

2021-04-03

Here is a reworked general pipeline for the program:
blast → accessions + scores → tax ids + scores → LCA → #reads per node / read → node mapping

3.4 Things to do if time is left

- Write parser for more blast formats.
- Include GTDB data for archaea.

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

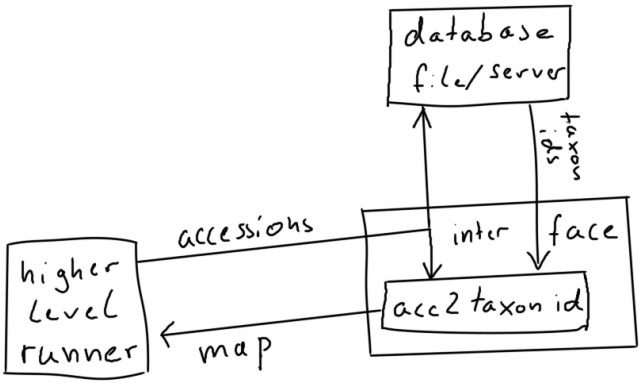

Figure 3: General idea of megan_map.db interface.

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 three 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!

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.

4.4 Feedback & Adjustments

2021-03-15

I got no negative feedback here, so I just added some errors and I think task 1 is all done. Maybe something will come up during the next meeting at the end of the month.

2021-03-17

I just figured out that my module should be able to return the accession to key mapping for any key, not just NCBI taxonomy. I will have to slightly adjust the implementation for that.

2021-04-18

Fixed the crash when selecting a single key with an `IN` statement. Python converts a tuple to a string like this: `str(tuple([1, 2, 3])) -> '(1, 2, 3)'`, but unfortunately Python hates single tuples and converts single tuples like this: `str(tuple([1])) -> '(1,)'`. The trailing comma messes up the sqlite query, so I had the choice of manually removing that comma somehow or just using `==` instead of `IN`. Now I have to decide in an if statement whether there is only 1 accession or multiple. I will check if there is a more efficient way later. In order to check the length, I had to change the type constraints from `Iterable` to `List` though, because `Iterable` does not implement `Sized`.

Bad news: looking up roughly 10 000 000 accessions takes about 15-20 minutes. I will have to find a better way to fetch the taxonomy ids. For now I extended the code by another API which just returns a list of taxonomy ids instead of a full mapping.

5 Task 2: Parse Taxonomy Tree

5.1 Framework

2021-03-15

For the next task I will have to implement a bunch of parsers for NCBI, GTDB and Silva taxonomy, which produce a phylogenetic tree. My initial idea was to only parse the data on the first run and otherwise save a blank phylogenetic tree, but if I have to be flexible with the data source, I'm not sure that's still a good idea. I would have to maintain a bunch of blank trees, one for each source, which doesn't sound very elegant. For the phylogenetic tree I suppose it's a graph with an arbitrary amount of children per node, each node only has one parent and there is a root or a set of roots. I think the node id can just be the taxonomy id? Maybe each node can store its parent as I'm building the tree, that would make the LCA application easier if I stored depths too.

5.2 Taxonomy Data

2021-03-15

NCBI Has a *nodes.dmp* file which contains the nodes with data: tax_id, parent tax_id, rank and other. *names.dmp* contains a mapping of tax_id to name_txt and unique name, which sounds pretty useful too. With this data I could make a phylogenetic tree where each node has a pointer to its parent and depth (rank) just by parsing without any preprocessing.

GTDB I'm honestly not entirely sure what I'm looking at in the *bac120_taxonomy.tsv* and *ar122_taxonomy.tsv* files. According to *FILE_DESCRIPTIONS* it's *GTDB taxonomy for all bacterial genomes assigned to a GTDB species cluster*, but there is no description for what each column means. I suppose the first column contains the taxonomy id. Then there are *bac120.tree* and *ar122.tree* files, which contain a tree in Newick format each, but the format seems a bit odd. There is some sort of 'port' after each taxonomy id and closed bracket, but the file description doesn't really explain it. I'll have to ask about this immediately.

Silva Looks like there is a file that contains all relevant taxonomy data: a taxonomy id and the 'path' to this entry. Then there is also a tree in (proper) Newick format and another file with a map from taxonomy id to name. How ever, the big file and the tree/map files contain different entries. Also there exists a lsu and a ssu version of each file and I'm not sure what the difference is. According to the README the tree/map files are compatible with MEGAN, which I will take as a pretty big hint for which files to use.

I compared some data from Silva and NCBI and they seems to vastly differ. Silva has 50909 entries, while NCBI has 2817222 and I'll just assume the ids don't map to the same names. The question is: is this covered in the megan map database? I'll just write an Email to Timo now and continue with the tree structure tomorrow. Silva and NCBI are mostly clear, but I'm not sure about GTDB yet.

2021-03-17

Alright I'd rather be implementing the phylogenetic tree right now, but I have to figure the taxon input out first. Timo replied to my questions thoroughly yesterday. Here is what I learned:

- Focus on NCBI first, worry about the other two later.
- The MEGAN src/resources contains NCBI/GTDB map and tre files.
- The ids in those files are compatible with megan_map.db.
- I should try to find version 132 of the Silva files, because there exists a mapping of Silva ids to NCBI ids for MEGAN for that version.

Other questions were answered too, but I think this is the information I need to progress. I found all the Silva maps and trees, but I'm not entirely sure how to read the Silva to NCBI taxon id maps. I need to ask that next. If I figure that out, I can build my tree from Silva and simply replace the ids with the corresponding NCBI ids. Next up are the issues with GTDB. Let's say I read a blast tab file and map the accessions. If I built my tree from GTDB I map the accessions to GTDB ids I suppose. I would have to adjust my megan map module to return either NCBI or GTDB ids for the accessions. Or should I take the GTDB tree and try to fill it with corresponding NCBI ids? If could try to use megan map, but accessions occasionally map to different NCBI/GTDB ids, so that's no use. Either way I have to build a Newick parser first and for that I need a phylogenetic tree structure, that can be built from a Newick tree.

5.3 Phylogenetic Tree

2021-03-18

Last night and this morning I implemented a bunch of stuff, which I was too lazy to plan out here beforehand, but should be documented regardless. First up: *Phylogenetic Tree*. As far as I understand it's a rose tree with node labels. My original idea was to make it from a directed graph, but edges are redundant at the moment and I don't think they will be necessary for my project at all. Leaving out edges actually saved quite a bit of time when parsing the NCBI tree (from 15 s down to 9 s). I have to keep performance in mind, because my tool will be compared to the Java tool. The question is: am I'm sacrificing too much flexibility? Not sure. Anyway here is the phylogenetic tree, I'm not going to bother with a graphic:

- **PhyloTree**: contains a pointer to its root node and a tax_id to node map
- **PhyloNode**: contains a tax_id, name, pointer to its parent and list of its children

This structure may be expanded to contain a depth for each node without any extra costs. It might have to be expanded in the future anyway, depending on how my future tasks change.

5.4 Newick Parser

2021-03-18

Now that I know what data I'm exactly dealing with, I was able to implement a Newick parser for the files that come with MEGAN. Their Newick format is `(1,2,(3,4)5)6`, which means all nodes are labelled with a number and don't have edge lengths. First I tried out a parser from the python package *newick*, but it took almost two minutes to parse the *ncbi.tre* and then another 25 seconds to parse it to my phylogenetic tree. All of the Newick pseudocodes I looked at use some kind of string operations or even regex to parse the tree and I can't imagine that being efficient for large trees, because strings are immutable and the operations will create billions of strings. I came up with an implementation that looks at each character exactly once. I have the advantage, that I know the exact input format and don't have to cover a bunch of cases. The idea as illustrated in fig. 4 is pretty simple. It looks at each character and puts it into a switch case:

- c is a **digit**: continue to build the tax_id. This case occurs the most often, so it comes first.
- c is a **comma**: concludes a node left of the comma. If the node hasn't been created yet, create it first of course. Stay on the same level, because a comma builds siblings.
- c is a **open bracket**: creates a new node. It's going to be a child of the current node, so go a level deeper by updating the parent/node pointers. tax_id is filled in later.
- c is a **close bracket**: concludes a node. Fill in the tax_id of a node and go a level higher by updating the parent/node pointers.

Entries into the tree are made by putting the node into the tree's dictionary. Each new node is added to its parent's children list. I wrote a bunch of test cases, but I don't know how I could possibly guarantee that the entire NCBI tree is built correctly. Worst case would be if I got entirely different results when comparing my tool to the Java implementation. Actually, I could write a reverse parser and check if it equals the input. That would be a bunch of extra effort, but at least it would provide closure. This algorithm may be extended to assign a depth to each node too for no extra cost. For each open bracket increment depth and for each close bracket decrement depth. May be useful later in the project.

Alright I just added a quick implementation of a phylogenetic tree to Newick parser and some unit tests and everything seems to work splendidly. I'm really happy.

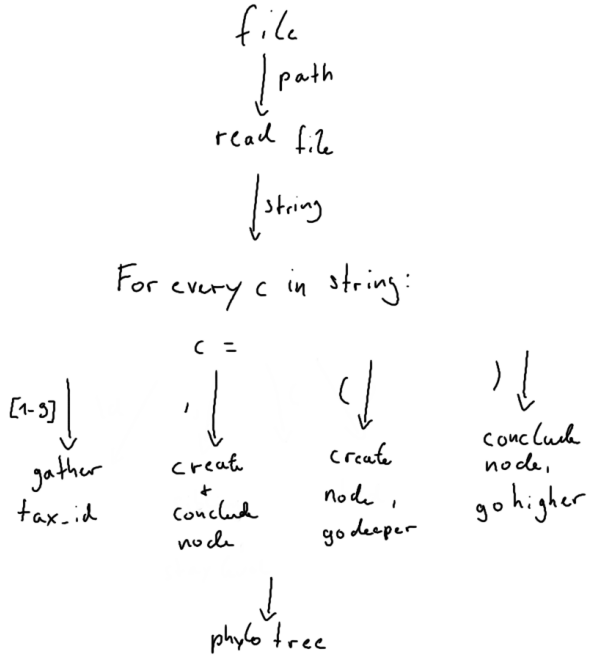

Figure 4: General idea of the Newick parser.

5.5 Map Parser

2021-03-18

The map files contain the taxonomy id, scientific name and then a bunch of other stuff separated by tabs in each line. The parsing is straight forward: read in the lines and extract the first two strings separated by tabs. Then use this matrix to fill in the scientific names for each node by accessing them with their tax_id. This should work for NCBI, GTDB and Silva the same way, but I might have to add a Silva to NCBI mapper later.

5.6 Feedback & Adjustments

2021-04-03

In the meeting on March 31 I got some good information. One thing I have to adjust for the map parser is also retrieving the *rank* of a taxon id. It's in the .map file too apparently in the last column. The rank should be included too, because it's part of a proper phylogenetic tree. I'm not actually sure if the rank is always in the last columns, some of the values look weird and I'm also missing a map of id to scientific name of ranks, but I can ask for that in the next mail. For now I can just do a provisional implementation.

Next up, I allowed to assume that any accessions I get as input can me mapped via megan map, which means, I can simply ignore Silva for now or at least don't have to worry about mapping Silva ids to NCBI ids. When I made my megan map interface compatible with GTDB, I made it useable with a generic key, so no adjustments necessary there.

Furthermore, I need to add a score or # reads or something field to my phylogenetic tree, so I can apply a *min support filter* later on.

2021-04-06

Got some more helpful information for this task, specifically the mapping of rank id to scientific rank name. Once I've added this to the code, the task should be complete.

6 Task 3: Parse Blast

6.1 Framework

2021-03-20

I'm not entirely done with task 2 yet, but I'll have to wait for next week with that, so I'm just going to start looking up some stuff for task 3 now. What am I trying to accomplish for this task? I receive a blast output file and want to extract information from them. A straight forward parser.

2021-04-03

Not much has changed, expect that the accessions are to be filtered by a top percent threshold. That means, per read I pick the accession with the highest score and disregard any accession in that read with less than x% of that score.

6.2 Blast output

2021-03-20

It's been a while since I've done anything with Blast files, I remember fasta much better, but I don't think I'll be needing fasta. I know there are different blast programs to use for whatever you're trying to align like DNA to DNA, DNA to protein, etc. but I'm not sure if they all have the same output format. I found a description of blast output and it looks like the user can customize it heavily with `-outfmt`. There are a bunch of default values, but they don't appear to contain accessions. Potential options I should probably be looking at are: *query accession* and *subject accession*. As I understand it, query is the input and subject is the reference, but does that mean query accession is what I'm looking for? I was also told that accessions may be filtered by their score, such that accessions below the threshold are simply discarded. *bitscore* is a default value, but there's also just *score*. bitscore seems to be more specific than raw score, so it's probably that. It would be nice to have a bunch of example files.

6.3 Diamond

2021-04-03

So Prof. Huson uploaded a bunch of 'small' 1 GB *DAA* files, which I can use as input for *Diamond* to receive blast files if I'm not mistaken. First, I have to download Diamond and learn how to use it. It appears, I can use `./diamond.exe view -a <daa file> -f 6 <options> -o <output file>` to generate blast files from daa files. *-f* signals the specification of the output format and *6* means blast tab (other blast options are: pairwise, xml). There are plenty of -f options to choose from, but I believe I'm interested in *sseqid* (Subject seq - id) and *bitscore* (Bit score). So far so good, but I don't understand how to separate reads yet. As I understood from the meeting, each file contains a number of reads and each read contains a number of accessions + scores. Maybe *sstart* (Start of alignment in subject) + *send*(End of alignment in subject), or *slen*(Subject sequence length), or *sallseqid*(All subject Seq - id(s))? I don't believe that'll work, but I have to try. None seem to have worked. Maybe I can separate the reads by their accessions? I have to ask, but first I should probably check if my 'accessions' are actually valid. Good news: the accessions from my output file appear to be mapped to taxon ids to some extent. Bad news: my megan map code is acting up. The mapping doesn't work when I call it on an accessions list with only 1 item. I'm not sure if that's actually relevant, I'll just ignore it for now, but keep it in mind.

2021-04-06

Timo helped me out with my confusion. I can include the *qseqid*(Query Seq - id) in my output file, which gives me the information on which read an accession belongs to. That makes a lot of sense and I probably should have figured that out myself. I tunnel-visioned on the wrong aspects. Anyway, here

is the final diamond call:

```
./diamond.exe view -a file.daa -f 6 qseqid sseqid bitscore -o out.txt
```

6.4 Blast Parser

2021-04-06

Now I have the 'read id' in the first column, accession plus '.1' in the second column and bit score in the last column. Should be easy enough to implement. I may add xml and pairwise blast format parsing later on. I'm just not sure what the most efficient implementation would be and if I can always assume that reads are continuous. If they are, it's going to be a lot easier. I'm also not sure whether I should store the read id somewhere or not. I shall bring that up when presenting my implementation.

2021-04-13

I am currently studying for my second to last bachelor exam, but I should not neglect this project for too long. The current implementation of the parser works as follows:

Read line by line (not all lines at once, because I don't want to load in 1 GB at once / redundantly create millions of objects) and check if I'm expanding a current read or starting a new read. If I'm starting a new read, filter the old one and collect all accessions that survived. Read ids and bit scores are discarded, only a list of surviving accessions per read (`List[List[str]]`) is returned. The parsing currently only works for the tab format, but may be expanded to xml and pairwise in the future.