# FastSubtrees

Giorgio Gonnella

Feb 27, 2023

# CONTENTS

CHAPTER

ONE

# FASTSUBTREES

*Fastsubtrees* is a Python library and a command line script, for handling fairly large trees (in the order of magnitude of millions nodes), in particular allowing the fast extraction of any subtree. The main application domain of *fastsubtrees* is working with the NCBI taxonomy tree, however the code is implemented in a generic way, so that other applications are possible.

The library functionality can be accessed both from inside Python code and from the provided command line tool `fastsubtrees`.

## 1.1 Introduction

For the use of *fastsubtrees*, nodes must be uniquely identified by non-negative IDs. Furthermore, the space of the IDs must be compact (i.e. the maximum ID should not be much larger than the number of IDs).

The first step when using *fastsubtrees* is to construct a tree representation. The operation requires a source of IDs of elements and their parents, which can be a tabular file, or any Python function yielding the IDs.

This operation just takes a few seconds, for a tree with million nodes, such as the NCBI taxonomy tree. It must be done only once, if a tree does not change, since the resulting data is stored to file.

The IDs of the NCBI taxonomy tree fullfill the conditions stated above. However, the library can be used for any tree. A way to use the library with IDs which do not fullfill the conditions, it to map them to an ID space which does, and store the original IDs as an attribute.

Besides the IDs, a tree can contain further information, e.g. integers, floats or other data, here called attributes, associated to the nodes. Each node can contain zero, one or more values for an attribute. To add values for an attribute, a tabular file or another data source (a Python function) is selected.

The data for any subtree can then be easily and efficently queried; thereby the node IDs and/other selected attributes can be retrieved.

The tree representation is dynamic, i.e. both the tree topology and the attribute values can be edited and changed.

## 1.2 Working with the library

### 1.2.1 Installation

The package can be installed using `pip install fastsubtrees`.

### 1.2.2 Command line interface

The command line tool `fastsubtrees` allows constructing and modifying a tree (subcommand `tree`), adding and editing attributes (subcommand `attribute`) and performing a subtree query (subcommand `query`).

The command line interface is further described in the CLI manual.

#### CLI example: working with the NCBI taxonomy tree

The example below uses the `fastsubtrees` command, as well as the `ntdownload` library (installed as a dependency, by `pip`) for obtaining the NCBI taxonomy data.

```
ntdownload ntdumps                                        # download NCBI taxonomy data
fastsubtrees tree nt.tree --ncbi ntdumps/nodes.dmp -f  # create the tree
fastsubtrees query nt.tree 562                            # query node 562

# attributes
ATTRTAB=data/accession_taxid_attribute.tsv.gz          # data file
TAXID=2; GENOME_SIZE=3; GC_CONTENT=4                      # column numbers, 1-based

fastsubtrees attribute nt.tree genome_size $ATTRTAB -e $TAXID -v $GENOME_SIZE -t int
fastsubtrees attribute nt.tree GC_content $ATTRTAB -e $TAXID -v $GC_CONTENT -t float

fastsubtrees query nt.tree 562 genome_size GC_content  # query including attributes

# taxonomy names
ntnames ntdumps >| names.tsv                              # prepare data from names dump
fastsubtrees attribute nt.tree taxname names.tsv       # add names as attribute
fastsubtrees query nt.tree 562 taxname genome_size     # query including taxa names
```

#### Using NtSubtree

The package `ntsubtree` (installable by `pip`) simplifies working with the NCBI taxonomy even more. Tree and the taxonomic names tables are automatically created and stored in a central location.

```
# first run after installing automatically downloads and constructs the tree

ntsubtree query 562               # taxonomic names displayed alongside the IDs
ntsubtree query -n "Escherichia"  # Query by taxonomic name

# attributes
ATTRTAB=data/accession_taxid_attribute.tsv.gz          # data file
TAXID=2; GENOME_SIZE=3; GC_CONTENT=4                      # column numbers

```

(continues on next page)

(continued from previous page)

```
ntsubtree attribute genome_size $ATTRTAB -e $TAXID -v $GENOME_SIZE
ntsubtree attribute GC_content $ATTRTAB -e $TAXID -v $GC_CONTENT
ntsubtree query -n "Escherichia" genome_size GC_content

# check if a newer version of the taxonomy data is available
# and update the tree if necessary, keeping the attribute values:
ntsubtree update
```

### 1.2.3 API

The library functionality can be also directly accessed in Python code using the API, which is documented in the API manual.

#### API example: working with the NCBI taxonomy tree

The example below uses the `fastsubtrees` command, as well as the `ntdownload` library (installed as a dependency, by `pip`) for obtaining the NCBI taxonomy data.

```
# download the NCBI taxonomy data
from ntdownload import Downloader
d = Downloader("ntdumpsdir")
has_downloaded = d.run()

from fastsubtrees import Tree
infile = "ntdumpsdir/nodes.dmp"
tree = Tree.construct_from_ncbi_dump(infile)     # create the tree
results = tree.subtree_ids(562)                   # retrieve subtree IDs

attrtab="data/accession_taxid_attribute.tsv.gz"         # data file
taxid_col=1; genome_size_col=2; gc_content_col=3        # column numbers, 0-based

tree.to_file("nt.tree")
tree.create_attribute_from_tabular("genome_size", attrtab, elem_field_num=taxid_col,
                                   attr_field_num=genome_size_col, casting_fn=int)
tree.create_attribute_from_tabular("GC_content", attrtab, elem_field_num=taxid_col,
                                   attr_field_num=gc_content_col, casting_fn=float)
results = tree.subtree_info(562, ["genome_size", "GC_content"])

# taxonomy names
from ntdownload import yield_scientific_names_from_dump as generator
tree.create_attribute("taxname", generator("ntdumpsdir"))
results = tree.subtree_info(562, ["taxname", "genome_size"])
```

### Using NtSubtree

The package `ntsubtree` (installable by `pip`) simplifies working with the NCBI taxonomy even more. Tree and the taxonomic names tables are automatically created and stored in a central location. The first time the library is included these operations are done automatically.

```
import ntsubtree

tree = ntsubtree.get_tree()
results = tree.subtree_ids(562)

taxid = ntsubtree.search_name("Escherichia")
results = tree.subtree_info(taxid, ["taxname"])

attrtab="data/accession_taxid_attribute.tsv.gz"         # data file
taxid_col=1; genome_size_col=2; gc_content_col=3        # column numbers, 0-based

tree.create_attribute_from_tabular("genome_size", attrtab, elem_field_num=taxid_col,
                                   attr_field_num=genome_size_col, casting_fn=int)
tree.create_attribute_from_tabular("GC_content", attrtab, elem_field_num=taxid_col,
                                   attr_field_num=gc_content_col, casting_fn=float)
results = tree.subtree_info(562, ["genome_size", "GC_content"])

# check if a newer version of the taxonomy data is available
# and update the tree if necessary, keeping the attribute values:
ntsubtree.update()
```

## 1.2.4 Docker

To try or test the package, it is possible to use `fastsubtrees` by employing the Docker image defined in `Dockerfile`. This does not require any external database installation and configuration.

```
# create a Docker image
docker build --tag "fastsubtrees" .

# create a container and run it
docker run -p 8050:8050 --detach --name fastsubtreesC fastsubtrees
# or, if it was already created and stopped, restart it using:
# docker start fastsubtreesC

# run the tests
docker exec fastsubtreesC tests

# run benchmarks
docker exec fastsubtreesC benchmarks

# run the example application
docker exec fastsubtreesC start-example-app
# now open it in the browser at https://0.0.0.0:8050
```

### 1.2.5 Tests

To run the test suite, you can use `pytest` (or `make tests`). The tests include tests of `fastsubtrees` and of the subpackage `ntmirror`. The latter are partly dependent on a database installation and configuration which must be given in `ntmirror/tests/config.yaml`; database-dependent tests are skipped if this configuration file is not provided.

The entire test suite can be also run from the Docker container, without further configuration, see above the *Docker* section.

### 1.2.6 Benchmarks

Benchmarks can be run using the shell scripts provided under `benchmarks`. These require data, which is downloaded from NCBI taxonomy and some pre-computed example data which is provided in the `data` subdirectory (genome sizes and GC content).

The benchmarks can be convienently run from the Docker container, without requiring a database installation and setup, see above the *Docker* section.

### 1.2.7 Example application: Genome attributes viewer

An interactive web application based on `fastsubtrees` was developed using *dash*. It allows to graphically display the distribution of values of attributes in subtrees of the NCBI taxonomic tree. It is a separate Python package, which can be installed using `pip`, and depends on *fastsubtrees*.

It can also be installed using the Docker image of *fastsubtrees* (see above in the *Docker* section).

For more information see also the `genomes-attributes-viewer/README.md` file.

#### Local installation and startup

To application can be installed using `pip install genomes_attributes_viewer` or from the `genomes_attributes_viewer` directory of the *fastsubtrees* repository.

To start the application, use the `genomes-attributes-viewer`. The first time this command is run, the application data are downloaded and prepared, taking a few seconds. Startup on subsequent starts does not require these operations and is thus faster.

### 1.2.8 Other subpackages

#### NtSubtree

NtSubtree is a library which automatically downloads the NCBI taxonomy dump and constructs the `fastsubtrees` data for it. It allows to easily keep the data up-to-date. It is a separate Python package, which can be installed using `pip`, and depends on *fastsubtrees*.

The `query` command of the NtSubtree CLI tool automatically display also taxonomic names, alongside the IDs in query and allow to perform queries by taxonomic name.

For more information see also the `ntsubtree/README.md` file.

#### ntdownload

When working with the NCBI taxonomy database, a local copy of the NCBI taxonomy dump can be obtained and kept up-to-date using the *ntdownload* package, which is located in the directory `ntdownload`. It is a separate Python package, which can be installed using `pip`, independently from *fastsubtrees*.

Please refer to the user manual of *ntdownload* located under `ntdownload/README.md` for more information.

#### ntmirror

A downloaded NCBI taxonomy database dump can be loaded to a local SQL database, using the package *ntmirror*, which is located in the directory `ntmirror`. It is a separate Python package, which can be installed using `pip`, independently from *fastsubtrees*.

It contains also a script to extract subtrees from the local database mirror using hierarchical SQL queries.

Please refer to the user manual of *ntmirror* located under `ntmirror/README.md` for more information.

### 1.2.9 Internals

For achieving an efficient running time and memory use, the nodes of the tree are represented compactly in deep-first traversal order. Subtrees are then extracted in O(s) time, where s is the size of the extracted subtree (i.e. not depending on the size of the whole tree).

The IDs must not necessarily be all consecutive (i.e. some "holes" may be present), but the largest node ID (*idmax*) should not be much larger than the total number of nodes, because the memory consumption is in *O(idmax)*.

For each attribute defined in a tree, a file is created, where the attribute values are stored. The attributes are also stored in the same deep-first traversal order as the tree IDs.

#### Tree construction algorithm

The tree construction algorithm used here is the following, where the input data consists of 2-tuples `(element_id, parent_id)` and the maximum node ID m is not much larger than the number of IDs n.

1. iteration over the input data to construct a table *P* of parents by ID, i.e. `P[element_id]=parent_id` if `element_id` is in the tree, and `P[element_id]=UNDEF` if not, where UNDEF is a special value. This requires *O(n)* steps for reading the IDs and *O(m)* steps for writing either the ID or the *UNDEF* value to *P*; since *m>=n*, the total time is in *O(m)*.
2. iteration over table *P* to construct a table *S* of subtree sizes by ID; for each element the tree is climbed to the root, to add the element to the counts of each ancestor. This operation requires *O(n*d)* time, where *d* is the height of the node, which is in average case much lower than *m* and *d=m* is the worst case.
3. iteration over each node ID to construct the list *D*, consisting of the depth first order of the nodes, and the table *C* of the coordinates of all nodes in the tree data, by id. For this operation, first the root is added to *D* and *C*, then for each other node *x* in *P*, the tree is climbed and nodes added to a stack until the next not-yet-added ancestor is found. The position where to add it this node is computed by the next free position in the subtree of its parent (which must have been already added, by definition, thus the next free position in its subtree is known). After this, the next stack element is added, until *x* is added. Although this operation also requires climbing the tree, it takes in total *O(n)* time, since each node is added only once.

### Parallelizing the tree construction

Currently the slowest step of the construction, detailed in the previous section, is the second, i.e. the computation of *S*. Since each node must be count in the subtree size of all its ancestors, there is no easy way to reduce the time from *O(n*h)*.

To parallelize this step, one divides the parents table into *t* slices, and assign each to a different sub-process (not thread, because of the GIL). Each sub-process would then count the subtree sizes in the slice only. A version implemented with a shared table and a lock was too slow, since access to the table was concurrent among the sub-processes. In the current version, instead, each sub-process makes a own subtree sizes *S'* table. The sub-processes *S'* tables are summed up after completion for obtaining the *S* table.

This option can be activated in the CLI using the `--processes P` option, or in the API setting the `nprocesses` argument of `Tree.construct` and related methods. Benchmarks show that the parallel version did not significantly improve the performance on constructing the NCBI taxonomy tree, likely because of the overhead of process starting, array *S'* initialization and summing up of all *S'* to *S* after completion.

## 1.3 Community guidelines

Contributions to the software are welcome. Please clone this repository and send a pull request on Github, to let the changes be integrated in the original repository.

In case of bugs and issues, please report them through the Github Issues page of the repository.

## 1.4 Documentation

The complete documentation of Fastsubtrees is available on ReadTheDocs at https://fastsubtrees.readthedocs.io/ in website and PDF format.

## 1.5 Licence



## 1.6 Acknowledgements

This software has been originally created in context of the DFG project GO 3192/1-1 “Automated characterization of microbial genomes and metagenomes by collection and verification of association rules”. The funders had no role in study design, data collection and analysis.

CHAPTER

TWO

# COMMAND LINE INTERFACE

The command line interface of the library consists in a script `fastsubtrees` that can be used to construct and modify a tree, add attributes to nodes, and query subtree IDs and attribute values.

The list of subcommands is displayed using `fastsubtrees --help`. Using `--help` after a subcommand (e.g. `fastsubtrees tree --help` displays the syntax and details of the subcommand.

The following subcommands are available:

```
tree         Create or modify a tree.
attribute    Create, modify or remove an attribute.
query        List node IDs and/or attributes in a subtree.
```

## 2.1 Tree construction

The subcommand `fastsubtrees tree` is used to construct the tree representation, from data consisting in node IDs and the corresponding parent IDs. The data can be obtained from a tabular file, or from a different data source.

### 2.1.1 Construction from a tabular file

If the IDs of the elements and their parents are contained in a tabular file, the filename is given as an argument to `fastsubtrees tree`, e.g.:

```
fastsubtrees tree my.tree
```

Details of the format can be specified using options. The separator is specified using `--separator` (default: tab), the columns containing the IDs using `--elementscol` and `--parentscol` (as 1-based column numbers, default: 1 and 2), and the prefix of comment/header lines using `--commentchar` (default: #).

When using `nodes.dmp` from the NCBI taxonomy tree dump, the preset `--ncbi` can be used. Example:

```
fastsubtrees my.tree --ncbi ntdumpsdir/nodes.dmp
```

### 2.1.2 Generalized tree construction

In the generalized tree construction mode, using the option `--module`, the path to a Python module is passed. The module defines a function, yielding the node and parent IDs. The default function name is `element_parent_ids` and can be changed using the option `--fn`.

All positional arguments given to the script are passed to the function. If they contain a =, they are passed as keyword arguments (unless the option `--nokeys` is used). Examples:

```
fastsubtrees tree my.tree --module my_module.py a b c
fastsubtrees tree my.tree --module my_module.py k1=v1 k2=v=2 x --fn myfn
fastsubtrees tree my.tree --module my_module.py k1=v1 k2=v=2 x --nokeys
```

This is the called function in the tree cases:

```
element_parent_ids("a", "b", "c")
myfn("x", k1="v1", k2="v=2")
element_parent_ids("k1=v1", "k2=v=2", "x")
```

A module implementing the described interface for reading from a tabular file is provided under `fastsubtrees/ids_modules/ids_from_tabular_file.py`.

## 2.2 Modifying an existing tree representation

Existing tree representations can be modified using `fastsubtrees tree` with the options `--update`, `--add` or `--delete`.

### 2.2.1 Updating or resetting a tree

If the option `--update` or `--reset` is provided, the tree is modified so to reflect the given source of IDs of elements and their parents (tabular file or Python function). The result is a tree, which is functionally equivalent to a new tree, constructed with the same data source.

The difference between the two is that `--update` edits the existing tree and attribute data, while `--reset` recomputes the tree data from scratch, after dumping the attribute values and reconstructing the attribute files with the dumped values afterwards.

The common advantage of using `--update` or `--reset` is that the attribute files are not lost. Generally the reset operation performs better, since tree creation is fast. The update operation can be faster, if the tree is large and is only slightly modified.

### 2.2.2 Adding leaf nodes or new subtrees

If the option `--add` is used, new elements are added to an existing tree. The elements must not yet be present in the tree and they must all be connected to a node already present in the tree or added in the same operation.

### 2.2.3 Removing leaf nodes or subtrees

If the option `--delete` is used, all remaining positional arguments of the script are IDs of nodes. If a node is a leaf node, it is removed from the tree. If it is an internal node, the entire subtree under that node is removed.

### 2.2.4 Attributes when editing a tree

If attribute have been defined, as described in the following section, the attribute files are automatically detected and modified too, when adding or deleting nodes.

If nodes have been added by `--add` or `--update`, new attribute values for those nodes can be added using `fastsubtrees attribute --add`, as explained below.

## 2.3 Tree attributes

The tree can contain further information, except the IDs, in the form of attributes. Attribute values can be integers, floats or strings. Not all nodes will necessarily have an attribute value associated with them. Some nodes can contain multiple values for an attribute. Attributes are managed by the subcommand `fastsubtrees attribute`.

### 2.3.1 Adding an attribute

To create a new attribute, a source of attribute values is passed to `fastsubtrees attribute`. Similar to the tree construction case, the source can be a tabular file, or a Python module, specifiying a function yielding node IDs and attribute values. The same node ID can appear multiple times, in which case the attribute values will all be stored, as a list.

By default, attribute values are stored as strings. The option `--type f` can be used to apply a function `f()` to each attribute value. The function can be either from the standard library (e.g. `int` or `float`, or, if `--module` is used, from that module.

#### Attribute values from a tabular file

To create an attribute from a tabular file, the filename is passed, e.g.

```
fastsubtrees attribute my.tree myattr value.tsv
```

Also in this case the format options can be used, for changing separator, comment character and specifying the columns containing the ID of the nodes (`--elementscol`) and attribute values (`--valuescol`).

#### Generalized source of attribute values

As a generalized attribute values source, the path to a Python module is passed, using the option `--module`. The module defines a function, yielding tuples `(node_ID, attribute_value)`. The default function name is `attribute_values` and can be changed using the option `--fn`.

All positional arguments given to the script are passed to the function. If they contain a =, they are passed as keyword arguments (unless the option `--nokeys` is used). Examples:

```
fastsubtrees attribute my.tree myattr --module my_module.py a b c
fastsubtrees attribute my.tree myattr --module my_module.py k1=v1 k2=v=2 x --fn myfn
fastsubtrees attribute my.tree myattr --module my_module.py k1=v1 k2=v=2 x --nokeys
```

This is the called function in the tree cases:

```
attribute_values("a", "b", "c")
myfn("x", k1="v1", k2="v=2")
attribute_values("k1=v1", "k2=v=2", "x")
```

An example module implementing the described interface is provided under `fastsubtrees/ids_modules/attrs_from_tabular_file.py`.

### 2.3.2 Listing defined attributes

To list the attributes that have been created, use `fastsubtrees attribute` with the option `--list`.

### 2.3.3 Editing attribute values

To add new values for an attribute, `fastsubtrees attribute` with the option `--add` is used. New values of the attributes for a node are appended to the existing ones. If the existing ones shall be replaced by the new ones, use the option `--replace` instead of `--add`.

To remove the values of an attribute for a list of given nodes, use `fastsubtrees attribute --delete` specifying the nodes. To remove an attribute completely, use `fastsubtrees attribute --delete` without specifying any node.

## 2.4 Subtree queries

The subcommand `fastsubtrees query` loads a tree representation from file and performs a subtree query to return a list of node IDs and/or attributes of the subtree under a given node.

To run the query, two parameters are required:

- `tree`: File containing the tree.
- `subtreeroot`: ID of the root of the subtree for which the IDs have to be queried

For query the values of an attribute in a subtree, the attribute names are passed as further arguments, after the subtree root argument. The output is tabular and a header line is output, which summarizes the content of each column.

To hide the node IDs when attributes are printed, use the option `--attributes-only`. In this case, only nodes for which some attribute value exists are shown, unless the option `--missing` is used.

CHAPTER

# THREE

# LIBRARY API

`Tree()` is the main class of this package. An instance of the class represents the tree information.

## 3.1 Construction of the tree representation

To construct the tree representation, information about the nodes of the tree and their parent node is required.

### 3.1.1 Constructing from a tabular file

`Tree.construct_from_tabular(filename, separator="\t", elem_field_num=0, parent_field_num=1)` allows the construction of a `Tree()` object from a tabular file. Header or comment lines starting with a `#` are ignored. The `elem_field_num` and `parent_field_num` parameters are 0-based field numbers of the fields (columns) in the tabular file containing, respectively, element IDs and the IDs of the element parents. Parents can be defined before or after their children.

If the tabular file is a NCBI Taxonomy dump nodes file, the method `Tree.construct_from_ncbi_dump(filename)` can be used, which sets the separator and field numbers appropriately.

### 3.1.2 Constructing from a different data source

For constructing the tree from a different data source (e.g. a database) the `Tree.construct(generator)` class method can be used. The paremeter shall be a generator, which yields pairs of values, that are the ID of each node and the corresponding parent node ID. Parents can be defined before or after their children.

## 3.2 Saving to and loading from file

The tree representations can be stored to file using the instance method `tree.to_file(filename)` and re-loaded from such a file using the class method `Tree.from_file(filename)`.

## 3.3 Modifying an existing tree representation

A tree representation (constructed from an input source or loaded from a file) can be modified, by adding or deleting leaf nodes or entire subtrees.

### 3.3.1 Adding nodes

For adding new nodes, the `tree.add_nodes(generator)` method is used. The paremeter shall be a generator, which yields pairs of values, that are the ID of each node and the corresponding parent node ID. If attributes have been defined (see below), the corresponding files will be automatically adapted.

The optional parameter `list_added` can be set to a list, to which the IDs of added nodes are appended. The optional parameter `total` can be set to the number of tuples yielded by the generator, for displaying a tqdm progress bar.

The method will raise an exception if the added nodes already existed in the tree, or if a parent node does not exist in the tree or in the nodes added in the same call to `add_nodes`.

### 3.3.2 Removing nodes

For removing a subtree, the `tree.delete_subtree(node_number)` method is used. If the provided ID is for a leaf tree, then only that node is removed. If it is an internal node, the node, and the entire subtree under it are removed. If attributes have been defined (see below), the corresponding files will be automatically adapted.

The optional parameter `list_deleted` can be set to a list, to which the IDs of the deleted nodes are appended.

The method will raise an exception if the `node_number` does not exist.

### 3.3.3 Moving a node

A node (except the root) can be moved, by detaching it from its parent and re-attaching it to a different parent node in the tree. For this operation the `tree.move_subtree(subtree_root, new_parent)` method is used. If attributes have been defined (see below), the corresponding files will be automatically adapted.

The method will raise an exception if the new parent node is not present in the tree.

### 3.3.4 Updating or resetting a tree

A tree can be updated to reflect the contents of a data source of tuples `(node_id, parent)`. Thereby all nodes which are present in the tree but not generated by the data source, will be removed, nodes with a changed parent will be moved, and nodes not present in the tree will be added. The method for updating a tree is `tree.update(generator)`. The method will fail if the root is changed, or if the tree data is invalid.

Updating can be advantageous over creating a new tree, in terms of performance, if the changes are not many. If a tree had several changes, a more efficient way is to recreate the tree data, using the `tree.reset(generator)` method. This is equivalent to creating a new tree, but the attribute values of nodes are kept, if the nodes are still present in the tree after the reset.

Both update and reset have a `_from_tabular` and a `_from_ncbi_dump` version of the method for working with tabular files and, respectively, the NCBI taxonomy nodes dump file.

## 3.4 Working with attributes

Attributes are stored and accessed from files. The filenames are automatically computed from the tree filename. If the tree object has been constructed but not saved yet to file, the filename can be set with `tree.set_filename`. If a tree topology changes, the attribute values files will be adapted accordingly.

### 3.4.1 Creating attributes

To create an attribute, a source of element IDs and associated attribute values is required. This can be a generator function which yields tuples `(element_ID, attribute_value)`, and is passed to the function `tree.create_attribute(attribute_name, generator)` or a tabular file, whose name is passed to `tree.create_attribute_from_tabular(attribute_name, tabular_filename)`. In both cases, a single-argument casting function can be passed as the argument `casting_fn`, which converts the values from the values source, e.g. from strings to another datatype.

### 3.4.2 Modifying attributes

To delete the value of an attribute for a list of nodes, the method `tree.delete_attribute_values(attribute_name, nodes_list)` is used. To append new values to some nodes, the new values must be passed as a dict in the form `{node_id: attr_values_list}` to the method `tree.append_attribute_values(attribute_name, new_values)`. This does not touch the existing values. To replace them, instead use `tree.replace_attribute_values(attribute_name, new_values)`. This replaces all values for the nodes in the dictionary and leaves the rest of the values intact.

It is also possible to directly edit the list of attribute values for some nodes. To this purpose a dictionary of the form `{node_id: attr_values_list}` is obtained using the method `attrvalues = tree.load_attribute_values(attribute_name)`. After loading, the dictionary entries are changed, by deleting, adding or modifying some values and finally passed to the method `tree.save_attribute_values(tree, attribute_name, attrvalues)`.

### 3.4.3 Checking if an attribute exists

To check if an attribute exists, the `tree.has_attribute(attribute_name)` method can be used. The list of all attributes is returned by `tree.list_attributes()`.

### 3.4.4 Destroying attributes

To remove an attribute completely, the `tree.destroy_attribute(attribute_name)` method is used. To remove all attributes, the `tree.destroy_all_attributes()` method is used.

## 3.5 Subtree queries

### 3.5.1 List of IDs of a subtree

The list of IDs of a subtree whose root is node `subtre_root` is obtained using the method: `tree.subtree_ids(subtree_root)`.

### 3.5.2 Attribute values in a subtree

Using the method `tree.query_attribute(subtree_root, attributes)` attribute values for the given subtree, for multiple attributes are obtained as a dictionary in the form `{'attribute_name': [values]}`. The optional parameter `show_stats` can be set to True, to compute and output some statistics to the logger info channel.

To associate the attribute names to IDs, and optionally collect further information, such as the IDs of the parents of each node, the method `tree.subtree_info(subtree_root, attributes)` can be used. Optional parameters can be set, to include the subtree sizes (debug information only, since they still contain deleted nodes), the parents (`include_parents` option). The dictionary keys for the subtree sizes, node ids and parent ids can be set using the `node_id_key`, `subtree_size_key` and `parent_key`.

## 3.6 Verbosity

For slow operations, such as constructing a new tree, optional progress bars (based on the tqdm libary) can be displayed. They are enabled by setting `fastsubtrees.PROGRESS_ENABLED` to `True` (by default the value is `False`).

The output verbosity can be controlled by setting the log level. By default the logger is disabled. Log messages can be activated by using `fastsubtrees.enable_logger("INFO")`. For debugging, additional messages can be displayed by using `fastsubtrees.enable_logger("DEBUG")`.

CHAPTER

FOUR

# NTSUBTREE

NtSubtree is a package based on Fastsubtrees which automatically downloads and installs the NCBI taxonomy tree during setup, and make it easier to work with taxonomy names.

## 4.1 Installation

It can be installed using `pip install ntsubtree` and automatically installs `fastsubtrees` on which it depends.

## 4.2 CLI

The CLI tool `ntsubtree` is provided by the package. The first time the tool is called from the command line, the package data (NCBI taxonomy tree) is downloaded from NCBI, the fastsubtrees tree data is constructed, as well as a table of taxonomy names.

If anything goes wrong during the automatic download and construction, use `ntsubtree update --cleanup` to repeat the process.

After that, it is possible to update the data to the newest NCBI taxonomy data, by running `ntsubtree update`. This only re-downloads the data and reconstruct the tree data, if newer data is available. Furthermore, it conserves any attribute data which have been added to the tree.

To add new attributes to the tree, `ntsubtree attribute` can be used. The usage is identical to `fastsubtrees attribute`, except that no tree filename is passed.

To query the tree, the `ntsubtree query` command is used. The usage is identical to `fastsubtrees query`, except that no tree filename is passed.

Taxonomic names are displayed automatically in the query results, unless the option `--no-taxname` is used. Furthermore, it is possible to query the tree by using a taxon name instead of a taxon ID as a subtree root, using the option `-n`.

### 4.2.1 Example usage

```
ntsubtree query 562               # taxonomic names displayed alongside the IDs
ntsubtree query -n "Escherichia"  # Query by taxonomic name

ntsubtree attribute myattr values.tsv
ntsubtree query 562 myattr

ntsubtree update
```

## 4.3 API

The first time that ntsubtree is imported, the package data (NCBI taxonomy tree) is downloaded from NCBI, the fastsubtrees tree data is constructed, as well as a table of taxonomy names. This can be triggered by `python -m ntsubtree`.

The `ntsubtree.update()` function can be used to check if new taxonomy data is avalaible at NCBI and, if so, download it and update the tree, without loosing existing attribute data.

Working with the tree is done using the *fastsubtrees* package API. The `Tree` object is obtained using `get_tree()`.

Besides the IDs, the `taxname` attribute is automatically available. Furthermore, the `ntsubtree.search_name(query)` function can be used to retrieve a taxon ID to pass to the *fastsubtrees* tree query methods.

### 4.3.1 Example usage

```
import ntsubtree

ids_in_subtree = ntsubtree.get_tree().subtree_ids(562)

taxid = ntsubtree.search_name("Escherichia")
subtree_info = tree.subtree_info(taxid, ["taxname"])

tree.create_attribute_from_tabular("myattr", "attr-tsv")
results = tree.subtree_info(562, ["taxname", "myattr"])

ntsubtree.update() # check for updates
```

CHAPTER

FIVE

# GENOMES ATTRIBUTES VIEWER

Genomes Attributes Viewer is an example application that is built on top of fastsubtrees to depict its usage. The application consists of a comparison page to compare the attributes like genome size or GC content for two or more organisms.

## 5.1 Installation

The example application can be installed using `pip install genome_attribute_viewer`.

## 5.2 Starting the application

To start the web application use the `genomes-attribute-viewer` script.

On first run, preparation steps are run, which take about 13 seconds (the NCBI Taxonomy dump files are downloaded from NCBI, a tree file for fastsubtrees is generated, along files which contains the scientific names of organisms and faststubrees attribute files for genome size and GC content of bacterial genomes using the provided example data).

On subsequent runs these steps are skipped.

The application is available, by default, at the URL http://0.0.0.0:8050/ . The hostname and port can be set using the options `--host` and `--port` respectively.

## 5.3 Usage

In the home page of the web application, the user can select an attribute, which can either be genome size or GC content from the `Select Attribute` dialog box. After selecting the attribute, one has to search for an organism using either its name or taxonomy id given dialog box. More organisms for comparison can be added by clicking the `Add Taxon` button. Finally to generate the graphs and compare two or more organisms, click the `Compare` button.

CHAPTER

SIX

# NTDOWNLOAD

NtDownload is a tool for downloading and keeping up-to-date a local version of the NCBI Taxonomy database dump files

After download it creates a timestamp file, so that the next time the download is repeated only if a newer version is available.

## 6.1 Installation

The software is distributed as a Python 3 package and can be installed using `pip install ntdownload`.

## 6.2 Command line interface

### 6.2.1 ntdownload

To download the dump files, use the `ntdownload` script. Thereby the output directory is passed as CLI argument. If it does not exist, it is created. The file is downloaded using FTP, unless this does not work or the option `--force-https` is used, in which case HTTPS is used.

The dump files archive is unpacked after download and deleted, unless the option `--no-unpack` is used.

If the option `--exitcode` is used, then the exit code of the script is 100 if no newer version of the dump files was found, and thus nothing was downloaded. Otherwise the exit code is always 0 (or 1 on error).

### 6.2.2 ntnames

The script `ntnames` is provided, which, after download using `ntdownload` can be used for creating a list of taxonomy IDs and scientific names, which can be used as attribute source file for fastsubtrees.

## 6.3 API

### 6.3.1 Downloader

To download the dump files, use the Downloader class:

```
from ntdownload import Downloader
d = Downloader(output_directory_name)
has_downloaded = d.run()
```

The output directory is created if it does not exist. The dump files archive is unpacked and the archive deleted, except if the option `unpack=False` is used.

The download protocol is FTP unless it is not working or the option `force_https=True` is used, in which case HTTPS is used.

The return value of `run()` is `True` if a dump file was downloaded, `False` if no newer version was available.

### 6.3.2 Scientific names iterator

The function `yield_scientific_names_from_dump(ntdumps_dir)` yields tuples in the form `(tax_id, scientific_name)` reading them from the `names.dmp` file.

## 6.4 Test suite

The test suite is run using `pytest`.

CHAPTER

SEVEN

# NTMIRROR

NtMirror is a tool for creating and keeping up-to-date a local mirror of the NCBI Taxonomy database.

## 7.1 Requirements

The software is distributed as a Python 3 package.

An installation by pip is only possible, if the `mariadb` module is installed (which requires installation of MariaDB and its C and Python connectors).

## 7.2 CLI

The CLI has been developed and tested exclusively using MariaDB as a RDBMS.

### 7.2.1 Loading into the database

The script `ntmirror-dbload` is used to load the dump files into the database. It takes as arguments the database username, its password, the database name and the path to the connection socket, followed by the directory where the dump files are located.

If the database tables do not exist, they are created. If dump files are found, they are loaded into the database and deleted. If no dump file is found, then nothing is done.

The exit code of the script is 0 on success and 1 if an error occurs. If the option `--exitcode` is used, and no dump file is found, the exit code of the script is 100 (instead of 0).

### 7.2.2 Subtree search using hierarchical SQL

To list the IDs of a subtree of the NCBI taxonomy tree, the `ntmirror-extract-subtree` script can be used. It takes as arguments the database username, its password, the database name and the path to the connection socket, followed by the subtree root ID.

### 7.2.3 Example usage

The following example uses the `ntdownload` package to download the dumps and loads them into the database and extracts a subtree using `ntmirror`.

```
ntdownload ntdumpsdir
ntmirror-dbload myuser mypass mydb /path/to/db.socket ntdumpsdir
ntmirror-extract-subtree myser mypass mydb /path/to/db.socket 562
```

## 7.3 API

### 7.3.1 Database setup

To create the database tables, a SqlAlchemy connection object is necessary. This is passed to the `dbschema.create(connection)` method, which creates the tables, if they do not exist yet.

### 7.3.2 Loading the data using MariaDB

In MariaDB, the database data loading is performed using the `mysql` library and not using SqlAlchemy, since the loading is faster.

The `dbloader_mysql.load_all(ntdumpsdir)` function is used, to which the path of the directory containing the dump files is passed, followed by the database hostname, database username, its password, the database name and the path to the connection socket.

If no dump files are found, nothing happens. Otherwise, the dump files are loaded into the database. The function returns an array of tuples `(filepfx, filepath)` for each dump file which was loaded into the database.

### 7.3.3 Example usage

The following example uses the `ntdownload` package to download the dumps and loads them into the database and extracts a subtree using `ntmirror`.

```
from ntdownload import Downloader
from ntmirror import dbschema, dbloader_mysql

# this assumes that the SqlAchemy connection is available
dbschema.create(connection)

d = Downloader("ntdumpsdir")
d.run()
dbloader_mysql.load_all("ntdumpsdir", dbhostname, dbusername, dbpassword,
                        path_to_db_socket)
```

### 7.3.4 Loading the data using another RDBMS

Database data loading using SqlAlchemy was also implemented, so that the package can be used with other RDBMS, although it is slower than using the `dbloader_mysql` module.

To upload the dump files into the database the `dbloader_sqlalchemy` module can be used. The database must implement a `LOAD DATA LOCAL INFILE` SQL command. The `dbloader_sqlalchemy` version of `load_all()` takes two arguments: the dump file directory and a SqlAlchemy connection object.

## 7.4 Tests

To run the test suite, a YAML configuration file for the database connection must be provided. The database name, database username and password, hostname and port, drivername and path to the connection socket file must be provided.

For example:

```
database: ntmirror_test
username: ntmirror_user
password: ntmirror_pass
host: localhost
port: 3306
socket: ntmirror.sock
drivername: "mysql+mysqldb"
```

Then the test suite is run using `pytest` (or `make tests`)