tetres

Release 1.0.0

Lars Berling

Aug 11, 2023

CONTENTS:

CHAPTER

ONE

WORKING WITH TIME TREES

Contents

1.1 The TimeTree class

A `TimeTree` can be initialized with a given newick string using the constructor and its format options. Additionally, a `TREE` object (*Classes for the c library*) is generated and saved in the `TimeTree` and used for efficient RNNI distance computations.

1.1.1 TimeTree attributes

Method	Description
`TimeTree.etree`	returns the `ete3.Tree` object
`TimeTree.ctree`	returns the respective `TREE` object
`len(TimeTree)`	returns the number of leaves of the `TimeTree`
`TimeTree.fp_distance(t)`	returns the findpath distance to another `TimeTree` *t*
`TimeTree.fp_path(t)`	returns a `TREE_LIST` object, allocated memory needs to be freed!
`TimeTree.get_newick(format)`	returns the *write()* function of the `ete3.Tree` with the specified *format*, defaults to *format=5*
`TimeTree.copy()`	returns a deep copy of the current `TimeTree`
`TimeTree.neighbours()`	returns a list of `TimeTree`'s containing all neighbours at distance *1*
`TimeTree.rank_neighbours()`	returns a list of `TimeTree`'s containing only neighbours one *rank move* away
`TimeTree.nni_neighbours()`	returns a list of `TimeTree`'s containing only neighbours one *NNI move* away
`TimeTree.nwk_to_cluster()`	computes the set of all clades present in the given `TimeTree`
`TimeTree.apply_new_taxa_map()`	applies a new taxa map (in form of a dictionary) to a `TimeTree`

This is an example of how to access the different attributes of a TimeTree object:

```
from tetres.trees.time_trees import TimeTree, free_tree_list


# Initialize a time tree from a newick string
tt = TimeTree("((1:3,5:3):1,(4:2,(3:1,2:1):1):2);")

tt.ctree  # the TREE class object

tt.etree  # the ete3.Tree object

len(tt)  # Number of leaves in the tree tt --> 5

tt.fp_distance(tt)  # Distance to another TimeTree --> 0

path = tt.fp_path(tt)  # A shortest path to another TimeTree --> []
free_tree_list(path)  # Allocated memory needs to be freed after usage

tt.get_newick()  # Returns the newick string in ete3 format=5

ttc = tt.copy()  # ttc contains a deep copy of the TimeTree tt

tt.neighbours()  # a list of TimeTree objects each at distance one to tt

tt.rank_neighbours()  # list of TimeTree obtained by doing all possible rank moves on tt

tt.nni_neighbours()  # list of TimeTree obtained by doing all possible NNI moves on tt

tt.nwk_to_cluster()  # returns set of all clades in the tree
```

(continues on next page)

(continued from previous page)

```
tt.apply_new_taxa_map(new_map, old_map)  # Will apply the new taxa map to the tree
```

1.1.2 ete3 functionalities

Via the `ete3.Tree` object the respective function of the `ete3` package are available for a `TimeTree` object. For example drawing and saving a tree to a file:

```
from tetres.trees.time_trees import TimeTree

tt = TimeTree("((1:3,5:3):1,(4:2,(3:1,2:1):1):2);")

# Automatically save the tree to a specific file_path location
tt.etree.render('file_path_string')

# Defining a layout to display internal node names in the plot
def my_layout(node):
    if node.is_leaf():
        # If terminal node, draws its name
        name_face = ete3.AttrFace("name")
    else:
        # If internal node, draws label with smaller font size
        name_face = ete3.AttrFace("name", fsize=10)
    # Adds the name face to the image at the preferred position
    ete3.faces.add_face_to_node(name_face, node, column=0, position="branch-right")

ts = ete3.TreeStyle()
ts.show_leaf_name = False
ts.layout_fn = my_layout
ts.show_branch_length = True
ts.show_scale = False

# Will open a separate plot window, which also allows interactive changes and saving the␣
↪image
tt.etree.show(tree_style=ts)
```

See the `ete3` documentation for more options.

1.2 The TimeTreeSet class

A `TimeTreeSet` is an iterable list of `TimeTree` objects, which is initialized with a nexus file (as returned by a BEAST2 analysis), hence it contains a taxa map.

Method	Description
`TimeTreeSet.map`	a dictionary conataining the taxa to integer translation from the nexus file
`TimeTreeSet.trees`	a list of `TimeTree` objects
`TimeTreeSet[i]`	returns the `TimeTree` at `TimeTreeSet.trees[i]`
`len(TimeTreeSet)`	returns the number of trees in the list `TimeTreeSet.trees`
`TimeTreeSet.fp_distance(i, j)`	returns the distances between the trees at postition i and j
`TimeTreeSet.fp_path(i, j)`	returns a shortest path (`TREE_LIST`) between the trees at postition i and j
`TimeTreeSet.copy()`	returns a copy of the list of :class:`TimeTree`s
`TimeTreeSet.get_common_clades()`	returns and computes the set of shared clades among all trees in the set
`TimeTreeSet.change_mapping(new_map)`	Will apply the given new taxa map to all trees in the set

1.2.1 Reading Trees

A TimeTreeSet object can be initialized with a path to a nexus file.

```
from tetres.trees.time_trees import TimeTreeSet, free_tree_list


# Initializing with a path to a nexus tree file
tts = TimeTreeSet("path_to_nexus_file.nex")

tts.map  # a dictionary keys:int and values:string(taxa)

tts.trees  # A list of TimeTree objects

for tree in tts:
    # tree is a TimeTree object
    ...
tts[0]  # trees are accessible via the index

len(tts)  # Returns the number of trees in the TimeTreeSet object

tts.fp_distance(i, j)  # Returns the distance between trees i and j
path = tts.fp_path(i, j)  # Returns a shortest path between trees i and j
free_tree_list(path)  # Allocated memory needs to be freed after usage
```

1.3 General Functions

A list of the functions available from the module 'tetres.trees.time_trees'.

Function	Description
`time_trees.neighbourhood(tree)`	returns a list of `TimeTree` objects containing the one-neighbours of tree
`time_trees.get_rank_neighbours(tree)`	returns a list of `TimeTree` objects containing the rank neighbours of tree
`time_trees.get_nni_neighbours(tree)`	returns a list of `TimeTree` objects containing the NNI neighbours of tree
`time_trees.read_nexus(file)`	returns a list of `TimeTree` objects contained in given the nexus file
`time_trees.get_mapping_dict(file)`	returns a `dictionary` containing the taxa to integer translation of the given file
`time_trees.findpath_distance(t1, t2)`	Computes the distance between t1 and t2, returns `int`
`time_trees.findpath_path(t1, t2)`	Computes the path between t1 and t2, returns `TREE_LIST`, after usage memory needs to be freed!

Note: Both functions `time_trees.findpath_distance(t1, t2)` and `time_trees.findpath_path(t1, t2)` can be called with t1 and t2 being either a `TREE`, `TimeTree` or `ete3.Tree`, both have to be the same type!

Note: When using `time_trees.findpath_path(t1, t2)` the c code is allocating memory to the returned object. This memory needs to be freed with the `time_trees.free_tree_list(tree_list)` function to avoid memory leaks, see more info below!

1.3.1 Working with findpath_path and c memory

When using the `time_trees.findpath_path(t1, t2)` implementation it is important to free the memory of the returned `TREE_LIST` object. When calling the function the package will also throw a UserWarning indicating this. Below are some examples of how to use the findpath_path implementation and the underlying class `TREE_LIST`.

```
from tetres.trees.time_trees import TimeTreeSet, free_tree_list

t1 = TimeTree()
t2 = TimeTree()

path = findpath_path(t1.ctree, t2.ctree)  # Will throw a UserWarning
free_tree_list(path)  # Free the memory allocated by c

# Calling findpath_path without the UserWarning being printed
with warnings.catch_warnings():
    # Ignores the 'Free memory' warning issued by findpath_path
    warnings.simplefilter("ignore")
    # All following calls do the same thing, but the memory is not being freed
    path = findpath_path(t1, t2)
    path = findpath_path(t1.ctree, t2.ctree)
```

(continues on next page)

(continued from previous page)

```
    path = findpath_path(t1.etree, t2.etree)

# Use the c code to free the memory
from ctypes import CDLL
from tetres.trees._ctrees import TREE_LIST
lib = CDLL(f".../tetres/trees/findpath.so")
lib.free_treelist.argtypes = [TREE_LIST]
lib.free_treelist(path)
```

1.4 Classes for the c library

These classes are found in the `_ctrees.py` module. The corresponding CDLL c library is generated from `findpath.c`.

1.4.1 NODE

- `parent`: index of the parent node (int, defaults to -1)
- `children[2]`: index of the two children ([int], defaults to [-1, -1])
- `time`: Time of the node (int, defaults to 0)

Note: The attribute `time` is currently not being used!

1.4.2 TREE

- `num_leaves`: Number of leaves in the tree (int)
- `tree`: Points to a `NODE` object (POINTER(`NODE`))
- `root_time`: Time of the root `Node` (int)

Note: The attribute `root_time` is currently not being used!

1.4.3 TREELIST

- `num_trees`: Number of trees in the list (int)
- `trees`: List of trees (POINTER(`TREE`))

1.5 Class converter functions

These are found in `_converter.py` and convert one tree type into the other. When converting a ctree to an ete3 Tree the branch lengths are discrete integers since the ctrees do not have a branch length annotation.

Function	Description
`_converter.ete3_to_ctree(tree)`	traverses an `ete3.Tree` and construct the correct `TREE`
`_converter.ctree_to_ete3(ctree)`	recursively traverses a `TREE` and generates an `ete3.Tree`

CHAPTER

TWO

SUMMARIZING TREES

Contents

2.1 The Centroid class

```
class tetres.summary.Centroid(variation="greedy_omp", n_cores=None, select='random',␣
↪start='FM', subsample_size=200,
            tree_log_file="", max_iterations=None)
```

This is used to setup a Centroid object which then takes a `TimeTreeSet` as input to compute the centroid summary tree.

2.1.1 Variation

The variation parameter of a `Centroid` has to be one in ["inc_sub", "greedy"] (TODO: Still WIP).

Variation	Description
Greedy	Computes a centroid via the greedy path and neighbourhood search. Only considering the tree with the most imporved SoS value in each iteration.
Inc_sub	Starts with a subsample of trees from the set, computes the greedy_omp centroid variant and adds more trees to the subsample until all trees are part of the sample.
Iter_sub	Starts with a subsample of trees from the set, computes the greedy_omp centroid variant and then resamples a new subset, using the previous centroid as the starting tree.
Separate	Only computes rank move neighbours if the tree contains all common clades of the tree set
Onlyone	Prefers either NNI or Rank moves and switches this if a local optimum is reached
update-with-one	Similar to the incsub variation, only one tree at a time is added to the subsample
Online	Mimicks an online approach where samples arrive one after another and the centroid is computed after each sample starting from the previous centroid

Greedy

```
tetres.summary.Centroid(variation="greedy_omp")  # default, using multiple processes in␣
↪c!
tetres.summary.Centroid(variation="greedy")  # pure python version
```

Inc_sub

The parameter subsample_size defines the size of the subsample of trees that is added each iteration. The parameter max_iterations defines the number of iterations, if it is None the regular break is defined whenever an iteration is not successful at improving the previous centroid. If it is an integer then it defines the number of iterations that will subsample, if it is 0 the start tree will be returned.

```
tetres.summary.Centroid(variation="inc_sub", subsample_size=500, max_iterations=None)
```

Iter_sub

The parameter subsample_size defines the size of the subsample of trees that is sampled each iteration. The parameter max_iterations defines the number of iterations, if it is None the regular break is defined whenever an iteration is not successful at improving the previous centroid. If it is an integer then it defines the number of iterations that will subsample, if it is 0 the start tree will be returned.

```
tetres.summary.Centroid(variation="iter_sub", subsample_size=500, max_iterations=None)
```

Separate

Will only use one move, current implementation is for NNI moves only, needs to be switched in source code (_variations.py, line 147).

```
tetres.summary.Centroid(variation="separate")
```

Onlyone

Will always do one move (starting with rank moves as of current implementation) and switch the move type whenever a local optimum is found.

```
tetres.summary.Centroid(variation="onlyone")
```

update-with-one

Similar to the inc-sub variation but only one new tree is added in each iteration.

```
tetres.summary.Centroid(variation="update_with_one")
```

Online

Mimicks an online approach where the trees arrive one by one in the given order.

```
tetres.summary.Centroid(variation="online")
```

2.1.2 Selecting a tree

This is only the case if multiple trees have the same SoS value. The defualt is random and the options are either random, first or last. The second two options are depending on the ordering which is dictated by the way the neighbourhood is computed.

2.1.3 Starting tree

There are the options to start with the last, the first or any given index of tree from the given tree set. The default option however is the sorted Frechet Mean tree (ref), see the doc on FM for more detail.

2.1.4 Subsample size

This is used by some variations and can be set to any integer number (default is 200). This number indicates the size of the subsample that the variation will use in its iterations. See the incsub or itersub variations

2.1.5 Maximal iterations

This is used to limit the number of iterations the iterative subsampling and increasing subsampling centroid versions are computing. If it is None (default) then the regular break points of those variations apply, otherwise it will only compute upto max_iteration many iterations.

2.1.6 Computing the SoS

The n_cores parameters defines the number of cores to use, if -1 all available cores are used (default).

2.1.7 Tree logfile

This option will write the trees of each centroid iteration to the given file path. This includes the actual centroid as the last tree. Can be used for further analysis.

Note that for incsub for example the tree is logged after an iteration on the subsample. This results in much smaller log files.

2.2 Annotation of a centroid

To keep the discrete ranks of a centroid use this annotation method. Each rank get assigned the average height of that rank in the given tree set, guaranteed to keep the same ranked tree after the annotation.

```
from tetres.summary.annotate_centroid import annotate_centroid
cen, sos = Centroid(variation="greedy").compute_centroid(my_tts)  # centroid of the␣
↪TimeTreeSet my_tts
annotated_cen = annotate_centroid(cen, my_tts)  # Annotation with the branch lengths␣
↪from the TimeTreeSet my_tts
# the annotated_cen is a TimeTree object for further use such as writing the newick to a␣
↪file
```

2.3 Frechet Mean

A version of Sturms algorithm adapted to the RNNI tree space, based on computing shortest paths with the findpath algorithm.

```
from tetres.summary.frechet_mean import frechet_mean, frechet_mean_sort
fm = frechet_mean(my_tts)  # random selection of trees
fm_sort = frechet_mean_sort(my_tts)  # trees are sorted from highest to lowest Sum of␣
↪squared distances
# The idea is that tres with low sum of squared distances are used in the end of the␣
↪algorithm to refine the tree
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search