Table of Contents

About Me

Basic Information

Name: Rahul Kumar
University: Indian Institute of Technology, Kanpur
Major: Aerospace Engineering
Email:rahuliitky14@gmail.com
Github:rahul-iitk
Timezone:IST (UTC +5:30)

Background work and Programming Skills

I am a second year student of Indian Institute of Technology Kanpur. I am pursuing a degree in **Aerospace Engineering**.Mathematics has always been my favourite along with which I keep learning from diversified subjects. Being proficient in C and Python, I am also skilled in Matlab, Octave and some other scripting languages.

The experience of using C has always helped in taking python to the next level. What captures my interest in the featured project is I can lend a good amount of my expertise of python and data structure over here, as well as learn through the challenges to add it to my knowledge bank.

The Project

The Aim and Motivation

ETE, being a pure python library, provides a swift way for people to use it because python doesn't need a compiler, so everything goes smoothly on the python interpreter. The basic

motivation of this project is to develop the searching capabilities in ETE tree structures, in particular, the aim is to develop a new ETE module that allows search querying large collections of trees using regular-expression-like-queries. Permitting this would extend the applications of this framework and will enable users to perform complex queries especially in the Phylogenomics field.

Ideas and Plan

My work can be divided into 2 phases:

Phase 1: Developing a vocabulary of patterns which permit regular-expression-like-queries

The syntax of the pattern should be a reasonable compromise between usability and functionality. Moreover, tree patterns should allow common operators and must have a basic language to permit user defined functions and filters.

First let us define the basic language of our vocabulary. We can add a number of filters to characterize a Phylo Tree node. I have divided the filters into 2 categories:

1. General filters:

   This includes the properties which are common to all nodes in the Phylogenetic trees. Some of the major attributes are described below:

- name: name of the node, len(name)will give the length
- dist: branch length distance to the parent node
- support: branch support for the node
- leaf: the node is a leaf
- root: the node is the root
- age: relative age of the node

Further there will be attributes of the node pointing to a list of nodes:

- leaves: list of all leaf nodes
- species: list of all species names
- dups: list of all duplication nodes
- specs: list of all speciation nodes
- gloss: list of all gene losses in nodes
- sister: list of all sisters of the node

Now if the user wants to use the length of some list attribute attr to define a query , he/she can use len(attr) or nattr. For example, to filter all nodes with length of the list of duplication nodes greater than equal to 3 under the current node, the syntax would be:

len(dups) >= 3 ⇔ ndups >= 3

2. Specific filters:

User will also be allowed to specify attributes particular to leaf nodes or the ones generated using NCBITaxa module of ete3.We can use the abbreviated version of these properties for convenience. Some of these include:

- sp: species name, stored in PhyloNode.speciesof the leaf node.
- taxid: NCBI taxid number.
- nl:named lineage, the NCBI lineage track using scientific names

Now, we will define common operators, which will be used alongside the language to define an element pattern. These element patterns will be further combined using logical operators to form a complete pattern. This pattern will define a node which will be written in Newick format to define the complete tree. I am proposing the following operators to my vocabulary of patterns:

```
@ = target node

LOGICAL OPERATORS
                  ||
OR       | , or ,  &&
AND      & , and ,
NOT      not , !
RELATIONAL OPERATORS

EQUALS   == , = , ~=
COMPARISON OPERATORS >= > < <= != ==

CUSTOM OPERATORS

contains(attr,value)
custom functions
function arguments = {}

PYTHON STANDARD OPERATORS

is, is not, in, not in
```

Custom functions allows to define a set of user functions that should be accepted in the pattern vocabulary. This can be achieved by passing a dictionary custom_variables (`function arguments`)at every call.

Finally, we need to convert the user supplied pattern to a Tree search pattern which makes sense to python eval function which we want to use later. The **highlighted** parts in the

above code represents non-standard python operators, so we need to either define them (blueones) or convert them to one (redones).

My developed search engine support would both YAML and Newick. The basic reason to do this is because YAML would be nice for complex patterns, but newick would also allow using command line with inline pattern searches. If a user enters the tree in YAML, this is a simple prototype which I could use to convert YAML into newick:

```
import yaml
from ete3 import PhyloTree,Tree
importre
...
class yaml_to_newick(Tree):
        def__init__(self,yml,**kargs):
                data=yaml.safe_load(yml)          #yaml converted into dictionary
                data=newick(data)        #manipulating the dictionary into newick format
                data=str(data)
                d=""
                fori inrange(len(data)):
                        ifdata[i]!="\\":
                                d=d+data[i]
                data=convert(d)
                data=format(data,format=8) #changing data into required format in
                newick printdata
        def newick(data):
                fori,key inenumerate(data.keys()):
                        iftype(data[key])==dict:
                                data[key]=newick(data[key])
                                data[key]=str(data[key])
                d=data
                data =dict((v,k)fork,v indata.iteritems()ifv[0]is'{')
                data.update(dict((k,v)fork,v ind.iteritems()ifv[0]isnot'{'))
                returndata
        def convert(data):
         data=data.replace("'{","(").replace("\"{","(").replace("}\"",")").replace("}'"
,")")
                data=data.replace("{","(").replace("): ",")").replace("}",");")
                returndata
        def format(data,format):
                ifformat==8:
                        data=data.replace("'name': ","").replace("'","")
                returndata
...
if__name__ =="__main__":
        ...
        yml="""
        root:
            branch1:
                name:A
                branch1-1:
                    name1:B
                    name2:E
            branch2:
                name:C
                branch2-1:
                    name1:D
                    name2:F
        """
                data =yaml_to_newick(yml,format=8) #converts yaml into newick format=8 ...
```

Running the above code would generate the following Newick pattern:

```
(((C, (name2: F, name1: D)branch2-1)branch2, ((name2: E, name1: B)branch1-1,A)branch1)root);
```

I will have to modify the code slightly to handle YAML patterns. I plan to use regular expression replacement techniques to get rid of 'name1' and 'name2' if needed.

I have fixed my input syntax for newick strings. For instance, we want to convert this user supplied pattern to a Tree search pattern:

```
(
nchildren > 2
,
Hsa in species
,
name = "hello" || name = "bye" && leaf ){length(name)
< 3 or name is "pasa"} and dist >= 0.5 ;
```

We follow the following steps:

1. First of all we need to identify all the attributes and prefix each of them with "@."
2. Replace square & curly braces with common brace. Plus, replace all "n" prefixed attributes using lenand "@" with "__target".
3. Define custom_functiondictionary for the customized functions.

   ```
   custom_functions = {"length":length}
   ```

4. Replace all non-standard operators with python standard operators. Also, define any operators if needed. For eg. I will evaluate nl = Primates by translating it to len(set(__target.named_lineage).intersection(["Primates"]))>0. One thing to note here is that evaluation of = depends upon the fact that nl(named_lineage) is a list.

Note:

- If it turns out quite difficult or inefficient to identify all attributes, the syntax in my input newick would prefix '@' with my attributes.

- However, there won't be any need to change syntax in if my input string is in YAML because I can easily prefix '@' during the dictionary manipulation in my prototype.

The final tree search tree pattern in Newick format would look like:

```
(
__target.len(children) > 2
,
Hsa in __target.species
,
__target.name    =    "hello"    or    __target.name    =    "bye"    and
__target.is_leaf()  )(length(__target.name)  <  3  or  __target.name  is
"pasa") and dist >= 0.5 ;
```

whose node patterns can be processed by python evalfunction for any target node.

Furthermore, it should also be noted that the user can easily extend the vocabulary using custom methods or operators. For example, consider the ~=operator mentioned earlier : name ~= seq\d+ would match node whose name is seq01,seq02,etc. If I could somehow convert this statement to bool(re.match("(^seq\d+)",__target.name)) using string manipulation, and feed it to the eval function, I'm done. This can be done by adding a separate function for conversion, keeping the code neat.

The magic we need to do here is that, for all translations we require for the eval function, we need to do them in pieces(by sequence of functions), so that the same functions can be recycled and used for something with which we plan to extend my vocabulary later with. For eg. we could have a general wrapper function which takes strings and whose slight modification can help wrap strings around each other in complex ways. We could use such functions to change nattr into len(attr), or for something with which we plan to extend my vocabulary later with.

Phase 2: Implementing the search engine

The aim in this phase will to implement the most optimal way to find matches. As of my current work plan, the matcher used to search will be recursive, i.e matching for a particular node and its children in the tree with the pattern will be done recursively. Now few improvements that I plan to make in this are:

1. **Reducing number of recursive calls:** The main idea is to invoke heuristics improvements so that our matcher visits only the necessary nodes to find an optimal solution. One idea could be to scan the search pattern before checking it with the nodes, and if we know that certain conditions could never be met, we could omit such nodes. Another possibility could be to assign some sort of priority to each node if possible along with A* algorithm and visit the nodes in that order. Instead of generating all possible solution branches, a heuristic selects branches more likely to produce outcomes than other branches. It is selective at each decision point, picking

branches that are more likely to produce solutions, and thus reducing the number of recursive calls.

2. **Parallelisation:** Putting simply, by parallelisation I mean to match our pattern with several trees simultaneously. Its implementation will greatly enhance performance to scan large collections of trees as the current database host millions of those. There could be two approaches to address this:

- *Multi-processing:* Unlike multi-threading, multi-processing allows one to use multiple cores of our processor nicely. It might be most sensible to use `multiprocessing.Pool` which produces a pool of worker processes based on the max number of cores available on your system, and then basically feeds tasks in as the cores become available. I have described a small prototype for the purpose below:

```
from multiprocessingimport Pool
...
deffind_matching_nodes(self,local_vars,matches_per_tree ,*args):
        i=0
         pool=Pool() #use all available cores, otherwise specify the number you want as
an argument (eg. my processor has 4 cores so I could add processes=4 as an argument)
        fortree inargs:
            try:

                    pool.apply_async(self.find_match, args= (args[i], local_vars, i, ) )
#doing it asynchronously we can move on to another task before the first finishes
            except:
                print"Error: unable to use multiple cores"
            i =i +1
        pool.close()
            pool.join()
```

It sure does make a copy of all data, still keeping 20-30 trees shouldn't be a problem unless the tree is too large. I can also explore the possibility of using ipython's `ipyparallel` during the implementation, which is similar to multiprocessing but can span multiple machines making it more efficient.

Algorithmic improvement: Although I haven't really found an improvement as of now, there could be possibilities. For example:

- While checking each children node recursively I am permutating the children in my pattern, and the matching each permutation of my pattern with children nodes. Suppose if number of children are '*n*', there are '*n!*' permutations to check for children of each node that is to matched. And if the tree is large, this could be slight of a problem. Here we can also see of heuristics, if it can mitigate the problem in most cases.

- We are using a kind of Depth first traversal(DFS) while verifying our patterns for each node. However, if trees have absurdly long branching patterns, there could be a possibility of stack overflow while using recursion. So what I am proposing is to implement our DFS iteratively instead of using recursion.

  Although, as of now for any iterative way I can think of or searched, to do DFS there always comes the need of using stack. Hence it still won't be memory efficient algorithmically. But still, point to be noted is that using recursion implies calling a function, which in Python has a larger impact than simply adding an entry to a list. So it still could be very well possible that the iterative implementation outperforms the recursive implementation in practice. Both solutions could be fine and the problem I raise may not be a problem at all because of bottlenecks somewhere else. Following is the prototype of the iterative version:

```
defis_match(self,node,local_vars=None):
   i=0
   status =self.constrain_match(node,local_vars)
   ifstatus andself.children:
      iflen(node.children)==len(self.children):
         s <-newstack p
         <-newstack
```

```
            count <-newstack
            visited=[]
            s.push(node)
            p.push(0)
            while(s isnotempty):
               current =s.pop()
               i=s.pop()
               if(current isinvisited):
                  continue
               visited.add(current)
               st =self.constrain_match(current,local_vars)
               status &=st
               ifstatus ==Falseori>=len(permutations(current.children)):
                  i=i+1
                  ifi>=len(permutations(current.children)):
                     x=count.pop()
                     """Pop from stack s 'x' number of times"""
                     """Pop from stack p 'x' number of times"""
                     status=True
                     i=0
                  c=0
               foreach node v inpermutations(current.children)[i]:
                  c++
                        s.push(v)
                        p.push(i)
                       count.push(c)
            ifstatus:
               break
         else:
            status=False
         returnstatus
```

Common works

I have planned to develop the Python API based usage for my search ETE feature along with all of the phases. Additionally, I'll be writing tests side by side. Solving all the bugs in the last is a really bad idea as compared to not letting them emerge. However, if any hiccups still persists, I'll try to solve them before merging the final work.

Here is a basic prototype to test yaml_to_newickclass defined earlier:

```
...
importunittest
...

classconvenient(unittest.TestCase):
    # tests if yaml to newick conversion is okay
        deftest_conversion(self):
                yml="""
root:
    branch1:
        name: A
        branch1-1:
            name1: B
            name2: E
    branch2:
        name: C
        branch2-1:
            name1: D
            name2: F
"""
                result="""(((C,  (name2:  F,  name1:  D)branch2-1)branch2,  ((name2:  E,  name1:
B)branch1-1, A)branch1)root);"""

                self.assertEqual(yaml_to_newick(yml,format=8).newick,result)
classyaml_to_newick(Tree):
    ...
if__name__ =='__main__':
    unittest.main()
```

On running the above program, we get the following output confirming that the class works fine:

```
.
----------------------------------------------------------------------
Ran1test in0.002s

OK
```

Like other ETE features, after API is ready, creating a command line tool is simple too. It just calls a function and then the process proceeds. In setup.py one can see something like entry_points which define the command line tool, it would be ete_search.py in our case.
For instance, in case of visualisation part command will look something like:

```
ete3 --show --pattern TreePattern --target_tree MyTree.nw
```

Note: Tree pattern needs to be in newick format to use inline pattern searches.

How do I fit in

I have been involved in graph theory for 6 months now. Initially I have had some contributions to networkx, which is a Python package that deals with complex graph networks.You can take a look at my contributions:

- (NetworkX) Added requirements.txt#1885
- (NetworkX) Modified release.py#1888
- (ETE) Minor variable correction in tutorial_trees.rst#188
- (ETE) Few small corrections in tutorial related files #189

I have had a first year course in genetics, and hence I'm aware with the basic terminologies like speciation, duplication etc. And so I comfortably shifted to phylogenetic trees which are like the trees in graph theory.

My project will build tree search capabilities within the ETE module from scratch. I don't claim to have to have covered all the loose ends, given that part of the project is also to identify and implement the most efficient series class structure. But I do have a clear understanding of what the requirements are and have spent time in designing initial code structures. Also I have knowledge of some libraries that may be used like scikit,numpy. More than anything else, I want to read and learn new things, as and when they come up.

References

- Cluster Analysis: https://en.wikipedia.org/wiki/Cluster_analysis
- Pattern Recognition: https://en.wikipedia.org/wiki/Pattern_recognition
- Project idea: http://obf.github.io/GSoC/ideas/#tree-searching-using-regular-expression-like-queri es

- Source code: https://github.com/etetoolkit/ete
- Prototype: https://github.com/etetoolkit/treematcher
- Feature selection clustering & Spectral Feature Selection (SPEC): http://www.public.asu.edu/~jtang20/publication/FSClustering.pdf
- TreeCl: http://mbe.oxfordjournals.org/content/early/2016/03/22/molbev.msw038