Table of Contents

About Me

Basic Information

Name: Kartik Raj
University: Indian Institute of Technology, Kanpur
Major: Computer Science and Engineering
Email: kartik.rajjj@gmail.com
Blog: http://karrtikr.blogspot.in/
IRC: karrtikr
Github: karrtikr
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 **Computer Science and Engineering**. Mathematics has always been my favourite along with which I keep learning from diversified subjects. I use Ubuntu 15.04 as my primary work machine and Sublime text for development. Being proficient in C and Python, I am also skilled in Bash, Verilog, Octave, Sqlite and some other scripting languages.

I am familiar with the concept of version control and know how to use Git and Github.

Python has always been my first choice because of its flexible usability and designing. It has the ability to shrink lines of code to just one or two sentences. 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 specially in the Phylogenomics field.

Ideas and Plan

My work can be divided into 4 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.species` of 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 (blue ones) or convert them to one (red ones).

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
import re
...
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=""
		for i in range(len(data)):
			if data[i]!="\\":
				d=d+data[i]
		data=convert(d)
		data=format(data,format=8) #changing data into required format in newick
		print data

	def newick(data):
		for i,key in enumerate(data.keys()):
			if type(data[key])==dict:
				data[key]=newick(data[key])
				data[key]=str(data[key])
		d=data
		data = dict((v,k) for k,v in data.iteritems() if v[0] is '{')
		data.update(dict((k,v) for k,v in d.iteritems() if v[0] is not '{'))
		return data

	def convert(data):
	      data=data.replace("'{","(").replace("\"{","(").replace("}\"",")").replace("}'"
,")")
		data=data.replace("{","(").replace("): ",")").replace("}",");")
		return data

	def format(data,format):
		if format==8:
			data=data.replace("'name': ","").replace("'","")
		return data
...
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 `len` and "@" with "__target".
3. Define `custom_function` dictionary 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 `eval` function 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-threading:* We would be converting set of sequential instructions into a multi-threaded instruction which would utilize multiple processors simultaneously enhancing the performance many-folds. A simple prototype for parallel processing using `thread` module to match our pattern with several trees is shown:

```
def find_matching_nodes(self, local_vars, matches_per_tree , args):
        i = 0
        for tree in args:
            try:
                thread.start_new_thread(self.find_match, (args[i], local_vars, i) )
            except:
                print "Error: unable to start thread"
            i = i + 1
```

We can see if it is working if we add the following statement in `find_match` function:

```
print "Thread %s: %s" % (i , time.ctime(time.time()))
```

Output for 3 threads:

```
Thread 1: Tue Mar 22 11:00:57 2016
Thread 0: Tue Mar 22 11:00:57 2016
Thread 2: Tue Mar 22 11:00:57 2016
```

However, multi-threading in python is not known to be efficient. So I would most likely be using the next approach.

- *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 multiprocessing import Pool
...
def find_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)
        for tree in args:
            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.

3. **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:

```
def is_match(self, node, local_vars=None):
  i=0
  status = self.constrain_match(node, local_vars)
  if status and self.children:
    if len(node.children) == len(self.children):
      s <- new stack
      p <- new stack
      count <- new stack
      visited=[]
      s.push(node)
      p.push(0)
      while (s is not empty):
        current = s.pop()
        i=s.pop()
        if (current is in visited):
          continue
        visited.add(current)
        st = self.constrain_match(current, local_vars)
        status &= st
        if status == False or i>=len(permutations(current.children)):
          i=i+1
          if i>=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
        for each node v in permutations(current.children)[i]:
          c++
          s.push(v)
          p.push(i)
        count.push(c)
      if status:
        break
  else:
      status=False
  return status
```

Phase 3: Developing a way to auto generate patterns from a bunch of real trees

Once we have developed our vocabulary of patterns, it would be nice to be able to automatically generate patterns from group of trees which would enable one to find more trees of its kind. I have elaborated my ideas into the following points:

- First we need to identify the attributes that make a bunch of trees distinguishable from the others. I plan to do this by encoding tree and node properties properly and then further use simple machine learning techniques(described in the next points) to detect the attributes relevant to the purpose. I plan to use Matlab because it is quite easy to learn and start with. Moreover, I have an experience in octave which is very similar to it. I may also shift to python libraries like scikit-learn, numpy etc. to do all kinds of scientific computing if needed, because of their rich and powerful language features.

- One trivial algorithm to approach this phase, is by '*feature selection*'. We begin by considering each subset of our attributes. Each new subset is used to train a model, which is tested on a hold-out set. Counting the number of mistakes made on that hold-out set will give the error rate of the model and then we can choose the model with minimum error rate. Although this is computationally very intensive, it will provide the best performing feature(attribute) set for that particular type of model. I could use statistical techniques such as principal components analysis, but as per my current knowledge, the resulting features after *feature extraction* has taken place are of a different sort than the original features and may not easily be interpretable, so we will have to discuss and see to it.

  Note the last line refers to technique of '*feature extraction*'. Here new features are created which will give better results. But the problem is that the new features may be encoded and so it will be difficult to extract our user friendly pattern from it later.

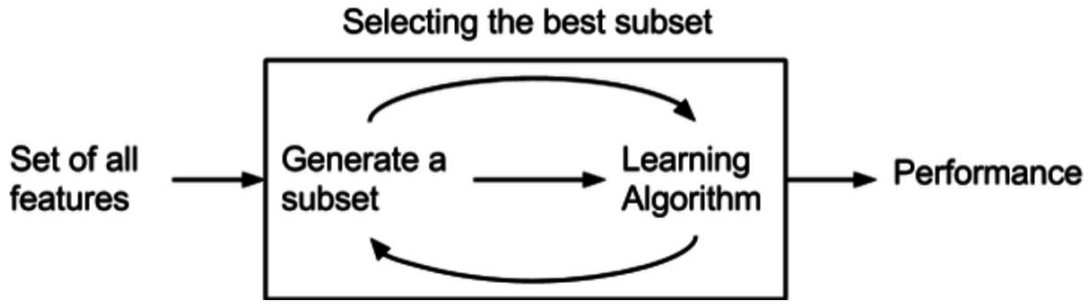

Fig: Trivial Algorithm(Wrapper method)

Another more practical approach to *feature selection* is by using '*feature weighting*'. Unlike assigning 0 or 1 to each feature like in *feature selection*, *feature weighting* assigns a value, usually in the interval [0,1] or [-1,1], to each feature. The greater this value is, the more salient the feature will be. The number of features to be picked is decided later by the learning algorithm.

- We could be using Spectral Feature selection (SPEC) algorithm as our learning algorithm for feature selection. Motivated by graph theory that states that graph structure information can be captured from its spectrum, SPEC studies how to select features according to the structure of the graph G. The translation for samples into matrices are given as:

$S_{ij} = e^{-\frac{||x_i - x_j||^2}{2\sigma^2}}$ where x_i and x_j are the i^{th} and j^{th} samples respectively. Putting simply, the weight of each feature f_i in SPEC is evaluated using three functions ψ_1, ψ_2, and ψ_3 which are derived from the spectrum of the graph. Each function ψ takes feature vector f_i and returns the final weight.

- One thing to note here is that the topology attribute is going to play a major role in deciding our pattern. We would want the topology to be the largest possible so our

tree family for the pattern can be made more distinguishable. So, we could give our learning technique a head start by giving priority(higher weight) to the topology attribute initially.

- Once those properties are detected, for instance, for a given collection of 20k trees, and a subclass of 100 trees, we would then learn and divide this collection of 20k trees into many groups by assigning a label to each tree(using clustering in unsupervised learning), and also have a rough pattern for each group. From as much as I tried and found out, treeCl looks promising and we could be using this module to implement clustering for large tree collections effectively. It clusters phylogenetic trees based on a matrix of between-tree distances.

- When we encounter a subclass(eg. of 100 trees), we would see in which groups it lies in, so that we can get the distinctive attributes of those 100 trees. If the trees lie in multiple groups, say 10 groups, the next thing we have to do in our algorithm is to first detect the "largest appropriate skeleton(topology) for our output pattern". We could use the rough pattern for each group to extract the final pattern. If our attributes doesn't fit appropriately to our skeleton, we will try the next most likely smaller skeleton for our bunch of trees.

The following prototype in combination with learning may help traversing to lower skeletons, although I need to construct the functionalities needed for the purpose:

```
class PatternExtractor(Tree):
    ...
    topology=self.get_topology([tree1,tree2,tree3])
    ...
    def get_topology(self,*args):
        skeletons=[]
        for tree in args:
            skeleton=get_skeleton(tree)
            #get_skeleton returns the topology for the tree, for eg. ((,),) for ((A,B),C)
            skeletons.append(skeleton)
        while """topology do not fit appropriately""":
            topology=next_structure(skeletons)
            #next_structure returns the next largest possible common substring ensuring the
topological structure
        return topology
```

Note: I have referred to skeleton as topology here as it would play the crucial role and should be the priority. I will add other attributes into the mix later to determine the exact pattern for my bunch of trees.

This would serve as our default attributes as long as user forces other conditions. We would want to make this technique more flexible by letting the user decide for which set of attributes he/she wants to generate patterns. For instance: use only topology, topology+node distances, etc. The declarative prototype will be like:

```
custom_attributes=["species","named_lineage"]
pattern=extract_pattern([tree1,tree2,tree3],default=True,custom_attributes)
#default takes True if the basis of commonalities detected should include default attributes
detected by our learning techniques, else False
```

Phase 4: Developing a visualization framework to highlight tree matches and differences

I plan to use QGraphicsScene to develop the visualization framework for my pattern. QGraphicsScene is a class which provides a surface for managing a large number of 2D graphical items. It has no visual appearance of its own; it only manages the items. It is generally used together with QGraphicsView for visualisation, however, in our case we'll be using ETE's tree drawing engine as our framework is supposed to be based on that.

PyQt4 is the Graphics framework module in python which contains all QGraphic classes required for the Qt4 drawing system. For starting to draw our trees we would first begin by inheriting the necessary QGraphicsScene classes in our class, initializing it, and then manipulating objects using mixed functions of QGraphics as well as our ETE's tree rendering engine to output our required display.

For this phase, as of now I have thought of the following visual outputs that the framework would give me:

- I am planning to create a face-to-face representation of two trees, where matching of the partitions can be highlighted.

- The framework will also permit the user to highlight what parts of a tree pattern are found in a given tree. It would have a legend for each pattern displayed.

- I would also have an option to display some common patterns that are found in the given tree. For instance I could display the 'rough patterns' allocated to each group in clustering I referred above, or I could display the gene losses which are common to find.
- I have thought the display of my trees to be circular by default as it's convenient to represent large trees by this. I would give an option customise things such that it is able to utilise aspects of trees like treestyles, nodestyles, node faces etc so things can

become more flexible for the user.

- I also plan to define layout functions in which can have some common layouts used for displaying patterns and phylogenetic trees. To make things more flexible, I will permit user to use their own custom layout functions in which he/she can use a mix of ETE's drawing engine as well as the QGraphicsScene functions.

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_newick` class defined earlier:

```
...
import unittest
...

class convenient(unittest.TestCase):

    # tests if yaml to newick conversion is okay
       def test_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)

class yaml_to_newick(Tree):
    ...

if __name__ == '__main__':
    unittest.main()
```

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

```
.
----------------------------------------------------------------------
Ran 1 test in 0.002s

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.

Timeline(Tentative)

Community Bonding Period (April 22 - May 22)

Goal: Community Bonding

- The principle focus in this period would be studying the ETE framework in detail, making notes on the things, which I can further extend my search engine with.
- I'll ask guidance from my mentor and exactly fix the syntax for my patterns so that I don't have much hesitation implementing it later.
- I also plan to at least start the basic coding in this phase itself, so that I can have a learning experience while moving into the next stages.

Week 1 - Week 2 (23 May - 5 June)

Goal: Developing the full vocabulary for my patterns

- I'll be writing code to convert user's input string into a pattern that can be evaluated by the `eval` function used in the prototype.
- I'll start writing the documentation of my vocabulary for patterns along with the code.

Week 3 - Week 4 (6 June - 19 June)

Goal: Implementing the search engine

- I'll start first with implementing the basic search. I'll take jaime's work as my starting point.
- In this period, I'll try invoking some heuristic improvements.
- I'll also be testing my work side by side.

- I'll implement parallelisation and other algorithm improvements if any.
- Fix bugs if any

Week 5 - Week 7 (20 June - 10 July)

Goal: Implementing the way to auto generate patterns

- Not less to say this will be not be a easy task and I will require some expertise and guidance.
- I'll ask guidance of my mentor and will learn more on machine learning and data mining.

Mid term Evaluation

- I'll will follow the steps as described in `Phase 3` and would implement the improvements as directed by my mentor.
- I will writing my ETE module and all the tests side by side.

Week 8 - Week 9 (11 July - 24 July)

Goal: Developing the visualization framework

- I'll learn Qt4 drawing system in more detail.
- Will create the custom visualisation layout as described using ETE's rendering engine.

Week 10 - Week 11 (25 July - 8 August)

Goal: Command line and API based usage

- I will develop the Python API based usage for my search ETE feature.
- After API is ready, I will be creating the command line tool for my feature.
- I will be completing any pending tasks, if any.

Week 12 - Week 13 (9 August - 22 August)

Goal: Software documentation & testing

- I will be completing the tests and documentation for every class and function I have implemented.
- Buffer period
- The final software will be hosted on github by the end of this period.

Future Work - Continue working over the developed ETE module post gsoc too. I'll love to see future implementations on it.

I'll be writing my notes throughout the progress. Later on I'll write IPython notebook tutorials for the work I'll have done.

I have no major plans for summer. Contributing 40 - 50 hours a week will not be a problem as I am very excited about the open source work I have to do, and will do my best. My summer vacation starts on 29th April. Hence, I will start coding a month earlier than the GSoC coding period, effectively giving me 4 months to complete the project. My academic year would begin by July 23, after which I will be able to spend around 30-35 hours a week.

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-queries
- 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