20.181 Lecture 3

Contents

Phylogenetic trees

Input: (a multiple sequence alignment)

1. AATGC
2. TATGC
3. GGTGG
4. ACTCG

Output: tree, an abstract representation of the same data ((1,4),(2,3))

Overview of Approach

(in pseudocode)

```
for each possible tree:
      calculate the score of (tree,data)
return tree with BEST score
```

Possible trees

- how many trees are there?
  - how does the number of possible trees increase with the number of leaves... linearly? ...exponentially?
    - start with the simplest unrooted tree, it has three leaves
    - how many ways are there to add another leaf? there are 3 ways- by adding the new leaf attached to each of the 3 existing branches (ignore the center leaf for now because we want to stick to binary trees)
    - now there are 5 places to add a leaf to a 4-leaf tree

        - every time you put a new branch down, you gain 2 more places to put a new branch: one from splitting an existing branch into two parts, and one from the new branch itself
    - `f_trees(n) = f_trees(n-1) * (2n-5)`
    - `for n leaves, f_trees(n) = (2n-5)!! <- that double factorial sign means to skip every other number`
    - `f_trees(n=10)= 34*10^6 f_trees(n=50) = 2.7*10^76`
    - Enumerating trees is not possible, so we are going to look only at a small number of possible trees. We need a search strategy. And the optimal search strategy will depend on the nature of the problem you're looking at.

What about rooted trees... well, we can just pick one of the leaves to serve as a root. So the number of rooted trees with n leaves is just the number of unrooted trees with (n+1) leaves, and f_rooted(n) = (2n-3)!!

Trees in Python

- For each node, we need to store:
    1. names (and sequences?)
    2. pointers to its left and right subtrees (its "children")

Data Structure

- We're going to use a built-in dictionary as our data structure
    - example tree = ((a,b),c)

```
tree1 = {'name':'a','left':None,'right':None} #for the
"subtree" that consists of leaf a
tree2 = {'name':'internal','left':tree1 ...
```

- well, we *could* do that, referencing our dictionary defined above. OR we could just avoid naming all the variables, and nest the definition of tree1 inside the bigger tree (tree2 above)

```
tree2 =
{'name':'internal','left':{'name':'a','left':None,'right':N
one},'right':{'name':'b','left':None,'right':None}}
```

both approaches are equivalent.

Parsing Function

- functions for dealing with this sort of data structure will be recursive

```
def leaves(tree):
      if (tree['name'] != 'internal'):
      return [tree['name']] # very important that this
      returns a list
```

```
      return leaves(tree['left']) + leaves(tree['right']) #
      "+" concatenates lists
```

```
def tree2string(tree): #a function to print out your tree
in newick format

      print '(' + left + ',' + right + ')'
```

- You'll be writing functions like these on the next homework. This code will probably have to be modified slightly to work in the correct context. (Note: the second function here is just an outline, filling in the details is homework problem #2.)