“I wonder about Trees” – Robert Frost

Hey! How come no turtles in this tree?
Learning Goals
• Describe the parsimony principle
• Introduce method for enumerating all trees
Recursion on trees: \texttt{graft}

\begin{verbatim}

groodies = ('Q',
            ('R', (), ()),
            ('S',
             ('T', (), ()),
             ('U', (), ()))

utree = ('U',
          ('V', (), ()),
          ('W', (), ()))

>>> graft(groodies, utree)
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', ('V', (), ()), ('W', (), ()))))

>>> graft(groodies, ('X',(),()))
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', (), ())))
\end{verbatim}

- At most one leaf of primary_tree has the same name as the root of graft_tree.
- If there is one such match, the function returns a new tree that is identical to primary_tree but with that leaf in primary_tree replaced by the entire graft_tree.
- If there is no leaf in primary_tree that matches the name of the root of graft_tree, the function simply returns primary_tree.
- No internal node of the primary_tree will have a name that matches the root of the graft_tree.
def graft(primary_tree, graft_tree):
    """If primary_tree has a leaf whose name is the same as the root of
    graft_tree then we return a new tree identical to primary_tree
    except with that leaf replaced by graft_tree. Otherwise, we
    just return primary_tree."""
Trees and the parsimony principle

Data
1  CCAGT
2  GCACT
3  CCTGA
4  GCTGT
Trees and the parsimony principle

Data

1 CCAGT
2 GCACT
3 CCTGA
4 GCTGT

"Better", i.e. more parsimonious
Another general strategy for inferring phylogenies

- Generate all possible trees
- Pick the most parsimonious given some data
Generating all possible trees from the ground up

# species (leaves)  1  2  3  4
# possible trees     1  1  3   

\[
\begin{array}{c}
\text{1} \\
\rightarrow \\
\text{2} \\
\rightarrow \\
\text{3}
\end{array}
\]
Draw all possible trees that result from adding a species 4 to this tree.
Draw all possible trees that result from adding a species 4 to this tree.
A convention for naming internal/ancestral nodes

tree = ( 'Anc', (1,(),()) , (2,(),()) )
The `add_leaf` function

```python
>>> leaf = (3, (), ()
>>> tree = ('Anc', (1,(),()), (2,(),())
>>> add_leaf(leaf, tree)
[('Anc', (3, (), ()), ('Anc', (1, (), ())), (2, (), ()))
 (Anc', (Anc', (3, (), ()), (1, (), ()))), (2, (), ())),
 ('Anc', (1, (), ()), ('Anc', (3, (), ()), (2, (), ()))),
]
```

Which illustration does the last tuple tree correspond to?
The `add_leaf` function

```python
>>> leaf = (4, (), ()
>>> tree = ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ()))))
>>> add_leaf(leaf, tree)
```
```python
[('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ())))), ('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ())))), ('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ())))), ('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ())))), ('Anc', (4, (), ()), ('Anc', (3, (), ()), ('Anc', (1, (), ()), (2, (), ()))))
]```
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == ():  # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree]  # wrap it in a list!
General case: three steps at each node

```
\hspace{1cm}
\begin{align*}
\text{add_leaf(new_leaf, tree)}
\end{align*}
```
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))
General case: three steps at each node

```
1
2
3
```

```
add_leaf(new_leaf, tree)
```

```
1
2
3
4
```

```
1
2
3
4
```

```
1
2
3
4
4
```

```
1
2
3
4
4
```

```
1
2
3
4
4
```

```
1
2
3
3
```

```
1
2
3
3
```

```
1
2
3
3
```

```
1
2
3
3
```

```
outgroup
```

```
right
(bottom)
```

```
left
(top)
```
The `add_leaf` chop shop: right tree

http://bananajams.wordpress.com/
The `add_leaf` chop shop: left tree

```
     1
   /   \
 2     1
 /       \
3       2
```

\[ \Rightarrow \text{add}_\text{leaf}(\text{new}_\text{leaf}, \text{left}) \]

```
     1
   /   \
 2     4
 /       \
3       3
```

```
     1
   /   \
 2     4
 /       \
4       3
```

[Image: http://bananajams.wordpress.com/]

def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == ():  # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree]  # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))

        # recur to add new_leaf on branches of right subtree
        temp_right_trees = add_leaf(new_leaf, right)
        for temp_right_tree in temp_right_trees:
            new_tree = (anc, left, temp_right_tree)
            output_trees.append(new_tree)
def add_leaf(new_leaf, tree):
    """Returns a list of all possible trees that result from adding new_leaf to tree."""
    root, left, right = tree
    anc = "Anc"
    if left == (): # a leaf.
        new_tree = (anc, new_leaf, tree)
        return [new_tree] # wrap it in a list!
    else:
        output_trees = []

        # put new_leaf as outgroup
        output_trees.append((anc, new_leaf, tree))

        # recur to add new_leaf on branches of right subtree
        temp_right_trees = add_leaf(new_leaf, right)
        for temp_right_tree in temp_right_trees:
            new_tree = (anc, left, temp_right_tree)
            output_trees.append(new_tree)

        # recur to add new_leaf on branches of left subtree
        temp_left_trees = add_leaf(new_leaf, left)
        for temp_left_tree in temp_left_trees:
            new_tree = (anc, temp_left_tree, right)
            output_trees.append(new_tree)

    return output_trees
Demo!
def all_trees(leaf_names):
    """Given a list of species, returns a list of all possible tree topologies."""

>>> all_trees([1,2,3])
[('Anc', (1, (), ()), ('Anc', (2, (), ()), (3, (), ()))),
 ('Anc', ('Anc', (1, (), ()), (2, (), ())), (3, (), ())),
 ('Anc', (2, (), ()), ('Anc', (1, (), ()), (3, (), ())))]
all_trees([4, 3, 2, 1])
all_trees([3, 2, 1]) ← all_trees([4, 3, 2, 1])
all_trees([3, 2, 1]) → all_trees([4, 3, 2, 1])
all_trees([1])

all_trees([2, 1])

all_trees([3, 2, 1])

all_trees([4, 3, 2, 1])
all_trees([3, 2, 1])

all_trees([2, 1])

all_trees([1])

all_trees([4, 3, 2, 1])
all_trees([1])

all_trees([2, 1])

all_trees([3, 2, 1])

all_trees([4, 3, 2, 1])
all_trees([3, 2, 1])

all_trees([2, 1])

all_trees([1])

all_trees([4, 3, 2, 1])
One general strategy for inferring phylogenies

- Generate all possible trees
- Pick the most parsimonious given some data
The number of trees grows quickly...

<table>
<thead>
<tr>
<th># species (leaves)</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td># possible trees</td>
<td>1</td>
<td>105</td>
<td>34,459,425</td>
<td>2.13458 x 10^14</td>
<td>8.200795 x 10^21</td>
<td>1.192568 x 10^30</td>
</tr>
</tbody>
</table>
Programming motifs: all vs. all

protsA = ['PLLYK', 'QSTE', 'NITQIVG', 'INE', 'QVAEA', 'YMSA']
protsB = ['LAGADLEQ', 'LAL', 'EAMERY', 'ENLEL']

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d = {}
for pA in protsA:
    for pB in protsB:
        d[(pA,pB)] = memoAlignScore(pA, pB, -9, blosum62, {})
Programming motifs: running the gauntlet

rnas = ['AUGACGCAGUAGUCA', 'UAGACAGUA', 'AGGUACAUc'...]

- If no RNA has a fold score above 7, return False
- Otherwise return True

```python
for rna in rnas:
    if fold(rna) > 7:
        return True
return False
```
Programming motifs: finding extremz

dictionary = [
    "abdomen",
    "abdominal",
    "abduct",
    "abduction",
    "aberration",
    "abet",
    "abhor",
    "abhorrence",
    "abhorrent",
    "abide",
    "abiding",
    "ability",
    "abject",
    "ablaze",
    ...
]

def z(input):
    '''Count z's in a string'''
    counter = 0
    for symbol in input:
        if symbol == 'z':
            counter = counter + 1
    return counter

def extremz(words):
    '''Find and return the word with the most z's'''
    best_count = 0
    best_word = ""
    for word in words:
        count = z(word)
        if count > best_count:
            best_count = count
            best_word = word
    return best_word
Recursion on trees: `graft`

```python
import cmath

groodies = ("Q",
            ("R", (), ()),
            ("S",
             ("T", (), ()),
             ("U", (), ()))
            )

utree = ("U",
         ("V", (), ()),
         ("W", (), ())
         )

>>> graft(groodies, utree)
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', ('V', (), ()), ('W', (), ()))))

>>> graft(groodies, ('X',(), ()))
('Q', ('R', (), ()), ('S', ('T', (), ()), ('U', (), ())))
```

- **At most one leaf** of `primary_tree` has the same name as the root of `graft_tree`.
- If there is one such match, the function returns a new tree that is identical to `primary_tree` but with that leaf in `primary_tree` replaced by the entire `graft_tree`.
- If there is no leaf in `primary_tree` that matches the name of the root of `graft_tree`, the function simply returns `primary_tree`.
- No internal node of the `primary_tree` will have a name that matches the root of the `graft_tree`.
def graft(primary_tree, graft_tree):
    """If primary_tree has a leaf whose name is the same as the root of
    graft_tree then we return a new tree identical to primary_tree
    except with that leaf replaced by graft_tree. Otherwise, we
    just return primary_tree."""
def graft(primary_tree, graft_tree):
    """If primary_tree has a leaf whose name is the same as the root of
    graft_tree then we return a new tree identical to primary_tree
    except with that leaf replaced by graft_tree. Otherwise, we
    just return primary_tree.""

    root, left, right = primary_tree
    if root == graft_tree[0]:
        return graft_tree
    elif left == ()
        return primary_tree
    else:
        left_graft = graft(left, graft_tree)
        right_graft = graft(right, graft_tree)
        return (root, left_graft, right_graft)