Genetic Heritage

The following 7 strings were generated by taking an existing string and with a small probability either inserting a new character, deleting an existing character, or changing to a new character randomly. This created two child strings, each of these strings with around 0-20 characters changed from their parent. And from the two child strings, four grandchild strings were created, two from each child. The 4 grandchild strings also have random character changed from their parents. As result, we got 7 strings but unfortunately the order of the strings has been lost.

```
(a,'CAGCGGGTGCGTAATTTGGAGAAGTTATTCTGCAACGAAATCAATCCTGTTTCGTTAGCTTAC
(b,'CAAGTCGGGCGTATTGGAGAATATTTAAATCGGAAGATCATGTTACTATGCGTTAGCTCACGG
(c,'CATGGGTGCGTCGATTTTGGCAGTAAAGTGGAATCGTCAGATATCAATCCTGTTTCGTAGAAA
(d,'CAAGTCCGCGATAAATTGGAATATTTGTCAATCGGAATAGTCAACTTAGCTGGCGTTAGCTTTA
(e,'CAAGTCCGGCGTAATTGGAGAATATTTTGCAATCGGAAGATCAATCTTGTTAGCGTTAGCTTAC
(f,'CACGGGCTCCGCAATTTTGGGTCAAGTTGCATATCAGTCATCGACAATCAAACACTGTTTTGC
(g,'CACGGGTCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTAT
```

1. Longest common subsequence

Write python code to give the length of the longest common subsequence for two strings.

This implementation is recursive: the base case is two strings that can be either identical or not. If we just implement the algorithm recursively, the number of comparisons will be incredibly large very quickly. We can use dynamic programming to avoid this problem, since we will have to use the same pairs of strings multiple times.

In [1]:
```python
import numpy as np
# defining the genetic sequences as variables
a='CAGCGGGTGCGTAATTTGGAGAAGTTATTCTGCAACGAAATCAATCCTGTTTCGTTAGCTTACGGACTACGAC
b='CAAGTCGGGCGTATTGGAGAATATTTAAATCGGAAGATCATGTTACTATGCGTTAGCTCACGGACTGAAGAGG
c='CATGGGTGCGTCGATTTTGGCAGTAAAGTGGAATCGTCAGATATCAATCCTGTTTCGTAGAAAGGAGCTACCT
d='CAAGTCCGCGATAAATTGGAATATTTGTCAATCGGAATAGTCAACTTAGCTGGCGTTAGCTTTACGACTGACA
e='CAAGTCCGGCGTAATTGGAGAATATTTTGCAATCGGAAGATCAATCTTGTTAGCGTTAGCTTACGACTGACGA
f='CACGGGCTCCGCAATTTTGGGTCAAGTTGCATATCAGTCATCGACAATCAAACACTGTTTTGCGGTAGATAAG
g='CACGGGTCCGTCAATTTTGGAGTAAGTTGATATCGTCACGAAATCAATCCTGTTTCGGTAGTATAGGACTACG

def lcs(stra,strb):
    # make matrix (initialized to 0) that will be used to save the lengths
        #of the subsections
    # i and j are the start and end points of the subsection as the index
        #of the string
    m=[[0 for j in range(len(strb)+1)] for i in range(len(stra)+1)]
    for i in range(len(stra)):
        for j in range(len(strb)):
            if stra[i]==strb[j]: # if the letters are the same
                # 1 is added to the previous longest subsequence
                m[i+1][j+1]=m[i][j]+1
            else:
                m[i+1][j+1]=max(m[i+1][j],m[i][j+1])
    return np.amax(m)
```

2. Table of lengths of common subsequences

Generate the table of the lengths of the longest common subsequences for every pair of strings.

In [2]:
```python
lst=[a,b,c,d,e,f,g]
[[lcs(i,j) for i in lst] for j in lst]
```

Out[2]:
```
[[100, 74, 76, 73, 82, 84, 91],
 [74, 90, 67, 72, 80, 70, 71],
 [76, 67, 97, 65, 69, 81, 84],
 [73, 72, 65, 96, 81, 71, 69],
 [82, 80, 69, 81, 96, 74, 75],
 [84, 70, 81, 71, 74, 111, 97],
 [91, 71, 84, 69, 75, 97, 104]]
```

3. Relationship between strings

Manually examine the table, and infer the relationships between strings First, I want to find a node that is similar to two of the other nodes and less similar to 4 of the other nodes. This will be the grandparent.
Then I want to find 2 nodes that have 3 other nodes (parent and 2 children) that are similar and 3 other nodes that are less similar.
The 4 grandchildren, finally, will be most similar only to their parent and maybe a sibling and less similar to the other nodes.
Additionally, I want to consider the length of the strings as a minor factor that can help me make the decisions. If the probability of adding or taking away a letter are always the same, larger jumps

are less likely.
For simplicity, I first rewrite each of the columns in the table I produced with percentages, then group all the other strings into 2 or 3 categories: very similar, less similar, least similar, that describe groups of numbers, not necessarily a clear cutoff that is the same for every list, but something that makes sense for the case.

a
length: 100
{100,74,76,73,82,84,91}
more similar: g
less similar: f, e
least similar: b, c, d

b
length:90
{82,100,74,80,89,78,79}
more similar: e
less similar: a, d, f, g, c

c
length:97 {78,69,100,67,71,83,86}
more similar: a, f, g
less similar: b, c, d

d
length:96
{76, 75, 68, 100, 84, 74, 72}
more similar: e
less similar: a, b, c, f, g

e
length:96
{85, 83, 72, 84, 100, 77, 78}
more similar: a, b, d
less similar: c, f, g

f
length:111
{76, 63, 73, 64, 67, 100, 87}
more similar: g
less similar: a, c,
least similar: b, d, e

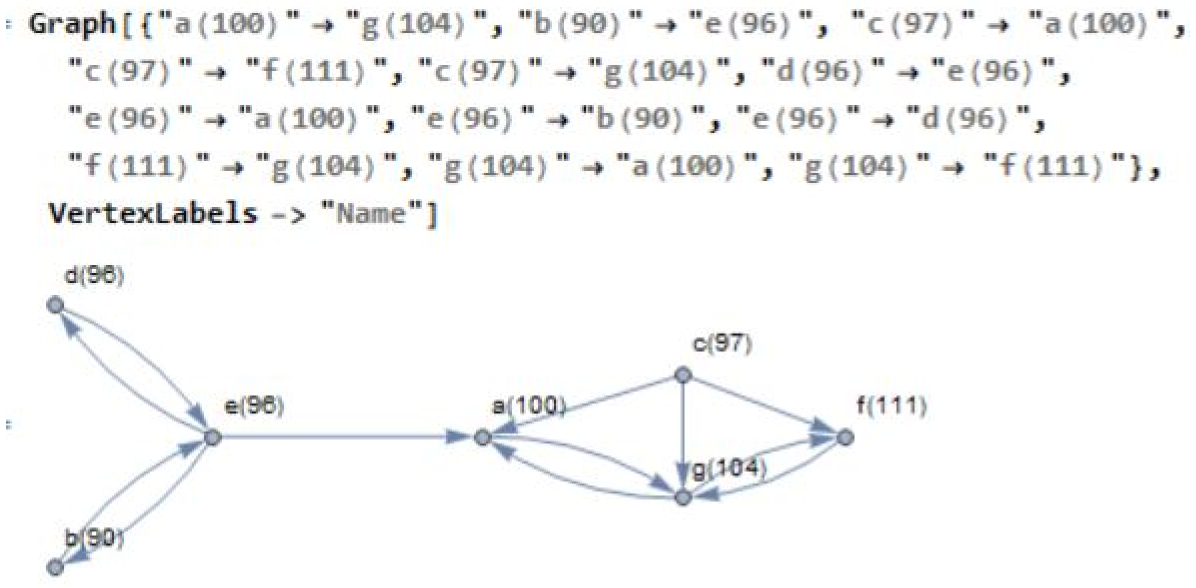
g
length:104

{88, 68, 81, 66, 72, 93, 100}
more similar: a, f, c
less similar: b, e

I then used Mathematica to plot a graph where every node is connected only to the most similar nodes.

In [3]:
```
from IPython.display import Image
Image("graph.jpg")
```

Out[3]:



```
Graph[{"a(100)" → "g(104)", "b(90)" → "e(96)", "c(97)" → "a(100)",
  "c(97)" → "f(111)", "c(97)" → "g(104)", "d(96)" → "e(96)",
  "e(96)" → "a(100)", "e(96)" → "b(90)", "e(96)" → "d(96)",
  "f(111)" → "g(104)", "g(104)" → "a(100)", "g(104)" → "f(111)"},
 VertexLabels -> "Name"]
```

The graph very nicely shows how e forms an arm with two children d and b. On the other side it is less clear, but since a has a direct connection to g and g to both a and f, I choose g as the second child of a and c as its grandchild. This is also supported by the lengths: 104 makes more sense as an intermediate step to 111. #1

4. Approach to find probabilities

How would you estimate the probabilities of mutation, insertions and deletions? (There might not be enough data to give meaningful estimates, but at least have a clear idea of the approach.)
First, I would isolate all the parent-children relationships and collect data on how the length and match changed:

Relationship	percentage change in length	percentage similarity to parent
a->e	-4%	82%
a->g	+4%	91%
e->d	0%	84%
e->b	-6.25%	83%
g->c	-6.7	81%
g->f	+6.7	93%

Let the variables representing the probabilities be s for switches, i for insertion and d for deletion. The table can give us a few raltionships between them. First, we can see the distribution of total change. 7-19% changed for the relationships we are observing. With more sample parent-child relationships, we can come up with an accurate average change.
$s + i + d = averagechange$
With more data points, we can also determine the difference between i and d, since
$i - d = percentagechanginlength.$
Once we have extracted all the information from the length of the longest common subsequence and the difference in total string length, we need to look at the strings themselves. If we have a parent string "abcdef" and a child that has abcdf as the longest common subsequence and the same length, it is important to know if the child is "abcdxf" or "abxcdf". In the first case, the letter e probably changed to x, in the second case e was removed and x was inserted between b and c. From this data, we can determine wether the operations were insertions or deletions and how often they occur.
To do this on a large scale, we can use what is called the Damerau-Levenshtein distance. Any of the calculations I suggested above assume that the resulting string is produced in the most efficient way. If the character A stays the same from one generation to the next, I assume it stayed and didn't get removed and replaced by a new A, etc. This means, if I align the strings above each other so constant letters are on top of each other, mutations are two different letters above each other, insertions are empty slots in the parent and deletion are empty slots in the child.
To get from Honey, to bee, for example, one would have to edit at least 4 characters. This is the levenshtein distance (Erickson, 2015)(Gilleland, n.d.)

1	**2**	**3**	**4**	**5**
H	O	N	E	Y
B	_	_	E	E

Doing the same kind of table for our strings can let us count the exact number of insertions (empty slots in the top string), deletions(empty slots in the bottom string) and mutations (changed slots). With a large enough sample, we can find the exact probability of each of the operations.
Below, I create a bottom-up dynamic programming solution for the Levenshtein distance, I then test it on the example "Honey" and "bee", which returns the correct editing distance 4 with 0 insertions, 2 deletions and 2 switches. I then add a few columns to the table above to find the Levenshtein distance between each of my parent-child pairs and the number of deletions and additions.

In [4]:
```python
import numpy as np

# takes two strings as input and returns their levenshtein distance,
# deletions, insertions, and switches
def levenshtein(a, b):
    # assign the size of the two strings to the length+1 for the matrix
    size_a = len(a) + 1
    size_b = len(b) + 1
    # create a matrix full of zeros
    matrix = np.zeros ((size_a, size_b))
    # initialize the outmost layers of the matrix
    # with numbers from 0 to the length of a and b
    for x in range(size_a):
        matrix[x,0] = x
    for y in range(size_b):
        matrix[0,y] = y
    #start the count for deletions, insertions, switches:
    deletions=0
    insertions=0
    switches=0
    # bottom-up fill the matrix with the required edits
    for x in range(1, size_a):
        for y in range(1, size_b):
            # if the two characters are the same
            if a[x-1] == b[y-1]:
                # choose above, to the left or diagonal
                #leaving the same if diagonal, otherwise +1
                matrix [x,y] = min(
                    matrix[x-1, y] + 1,
                    matrix[x-1, y-1],
                    matrix[x, y-1] + 1
                )
            else: # if the characters are not the same
                # choose above, to the left or diagonal, always adding 1
                matrix [x,y] = min(
                    matrix[x-1,y] + 1,
                    matrix[x-1,y-1] + 1,
                    matrix[x,y-1] + 1
                )
    # traversing the matrix to find the operations:
    i,j=len(a),len(b)
    while i!=0 or j!=0:
        if matrix[i-1,j-1]==min( matrix[i-1, j],
                                 matrix[i-1, j-1],
                                 matrix[i, j-1] ):
            if matrix[i,j]>matrix[i-1,j-1]:
                switches+=1
            i-=1
            j-=1
        elif matrix[i-1, j] == min( matrix[i-1, j],
                                    matrix[i-1, j-1],
                                    matrix[i, j-1] ):
            deletions+= 1
            i-=1
        elif matrix[i, j-1]== min( matrix[i-1, j],
                                   matrix[i-1, j-1],
                                        matrix[i, j-1] ):
                insertions+= 1
                j-=1

        # return the bottom right corner, which is the levenshtein distance,
        # as well as the deletions, insertions and switch operations
        return ([matrix[size_a- 1,size_b - 1],deletions, insertions,switches] )

levenshtein("honey","bee")
```

Out[4]:
```
[4.0, 2, 0, 2]
```

In [5]:
```
print(levenshtein(a,e))
print(levenshtein(a,g))
print(levenshtein(e,d))
print(levenshtein(e,b))
print(levenshtein(g,c))
print(levenshtein(g,f))
```

```
[26.0, 12, 8, 6]
[19.0, 3, 7, 9]
[23.0, 6, 6, 11]
[18.0, 8, 2, 8]
[23.0, 10, 3, 10]
[19.0, 5, 12, 2]
```

Relationship	percentage change in length	percentage similarity to parent	Levenshtein distance	deletions	insertion	switches
a->e	-4%	82%	26	12	8	6
a->g	+4%	91%	19	3	7	9
e->d	0%	84%	23	6	6	11
e->b	-6.25%	83%	18	8	2	8
g->c	-6.7	81%	23	10	3	10
g->f	+6.7	93%	19	5	12	2

From this data, we can find estimations of the probability so far, by adding the number of total input characters: $2 * 100 + 2 * 96 + 2 * 104 = 600$, the total number of deletions: $12 + 3 + 6 + 8 + 10 + 5 = 44$, insertions: $8 + 7 + 6 + 2 + 3 + 12 = 38$, and switches $6 + 9 + 11 + 8 + 10 + 2 = 46$.
We then get the probability of each of these operations for the sample:
$s = 46/600 = 0.076667$, so a 8% probability,
$i = 44/600 = 0.073333$, so a 7% probability, and
$d = 38/600 = 0.063333$, so a 6% probability.

Because of the low sample size, however, these results will likely change for a larger sample size, which is recommended.

5. General case

Can you devise an algorithm in the general case which might be able to infer such a tree of relationships? Give any strengths or weaknesses of your suggested algorithm.
Again, the Damerau-Levenshtein distance can be of use (as an improved measure of similarity over the longest common subsequence). Parents should have a shorter relative editing distance to their children than any other node. Thus, we connect each node with the group of either 1 or 3 closest relatives (they must be grouped together with respect to the other editing distances). Then we plot a graph connecting those likely related nodes and find a tree structure.
As we saw in part 3, this method has the weakness of random influences. Letter c, for example did not directly follow the pattern. We would also need to formulize a formal criterion for a group of editing distances to be closer related or not.

6. Complexity

Describe the complexity of your solution to identify related "genes" for this assignment. (Let M be the length of a gene, and N be the number of genes.) The algorithm I proposed would first find the editing distance from one gene to the other, which is an operation with time complexity $O(M^2)$, using a dynamic programming solution and two strings of length M since we need to build on the optimal solution in a M by M grid. This also means we have a space complexity of M by M.
We need to find this editing distance N! times to cover each possible combination of connected strings. This means the total complexity will be of order $O(NM^2)$

HC-tags:

1. #induction: I use each of the relationships as a piece of evidence towards a conclusion and deal with outliers to still come to a logical conclusion.
2. #algorithms: I come up with an efficient way to calculate the longest common subsequence and levenshtein distance and traverse the matrix to trace back the operations and find their probabilities.

Citations

Erickson, J. (2015, January 4). Algorithms Lecture series, Lecture 5: Dynamic Programming. Retrieved from http://jeffe.cs.illinois.edu/teaching/algorithms/all-algorithms.pdf (http://jeffe.cs.illinois.edu/teaching/algorithms/all-algorithms.pdf)
Gilleland, M. (n.d.). Levenshtein Distance. Retrieved December 11, 2018, from https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein (https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshteir

```
In [ ]:
```