Trying to re-create the fractal evolution of gene promoter networks using aggregation

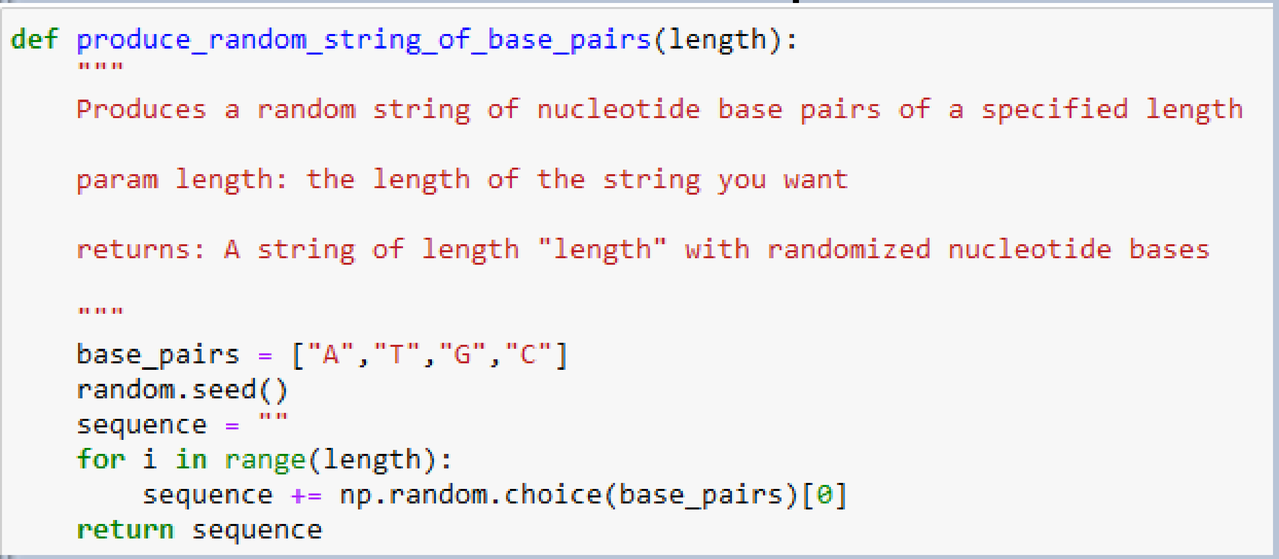
Mohit Srivastav

Using this paper by Preston R. Aldrich
Benedictine University, Lisle, IL 60532, USA

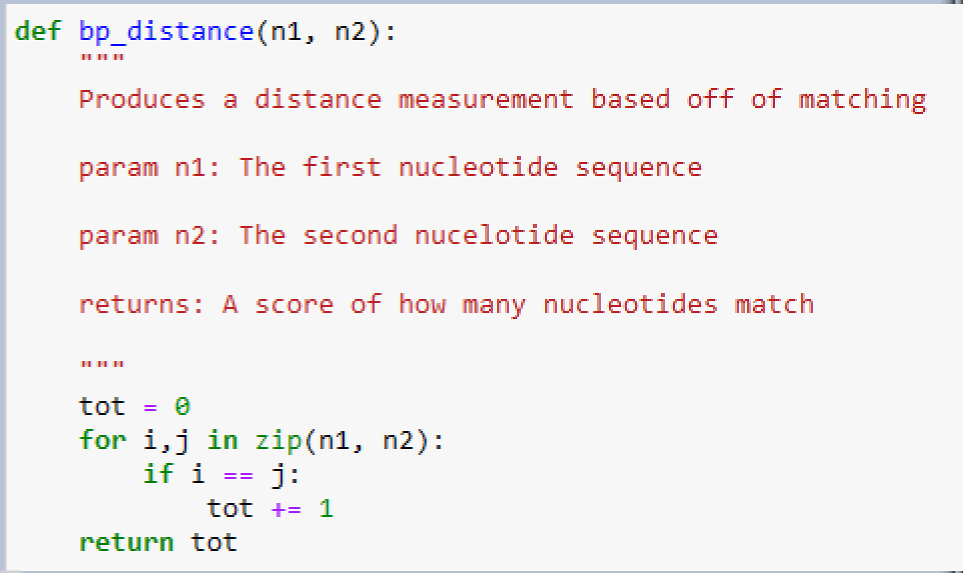
Initial Steps

- Produce a random “consensus candidate” that is the basis for your entire network. The length of this string is 11 base pairs.

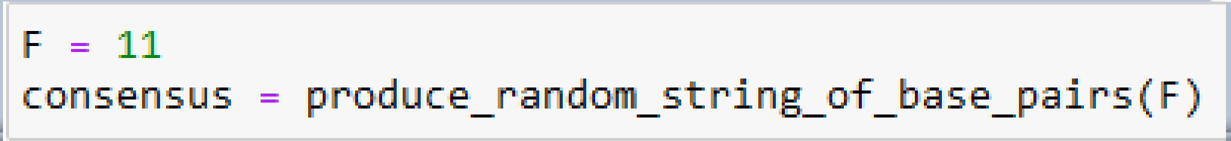
- The match score between two sets of base pairs is the number of nucleotides that match between sequences



```python
def produce_random_string_of_base_pairs(length):
    """
    Produces a random string of nucleotide base pairs of a specified length

    param length: the length of the string you want

    returns: A string of length "length" with randomized nucleotide bases

    """
    base_pairs = ["A","T","G","C"]
    random.seed()
    sequence = ""
    for i in range(length):
        sequence += np.random.choice(base_pairs)[0]
    return sequence
```



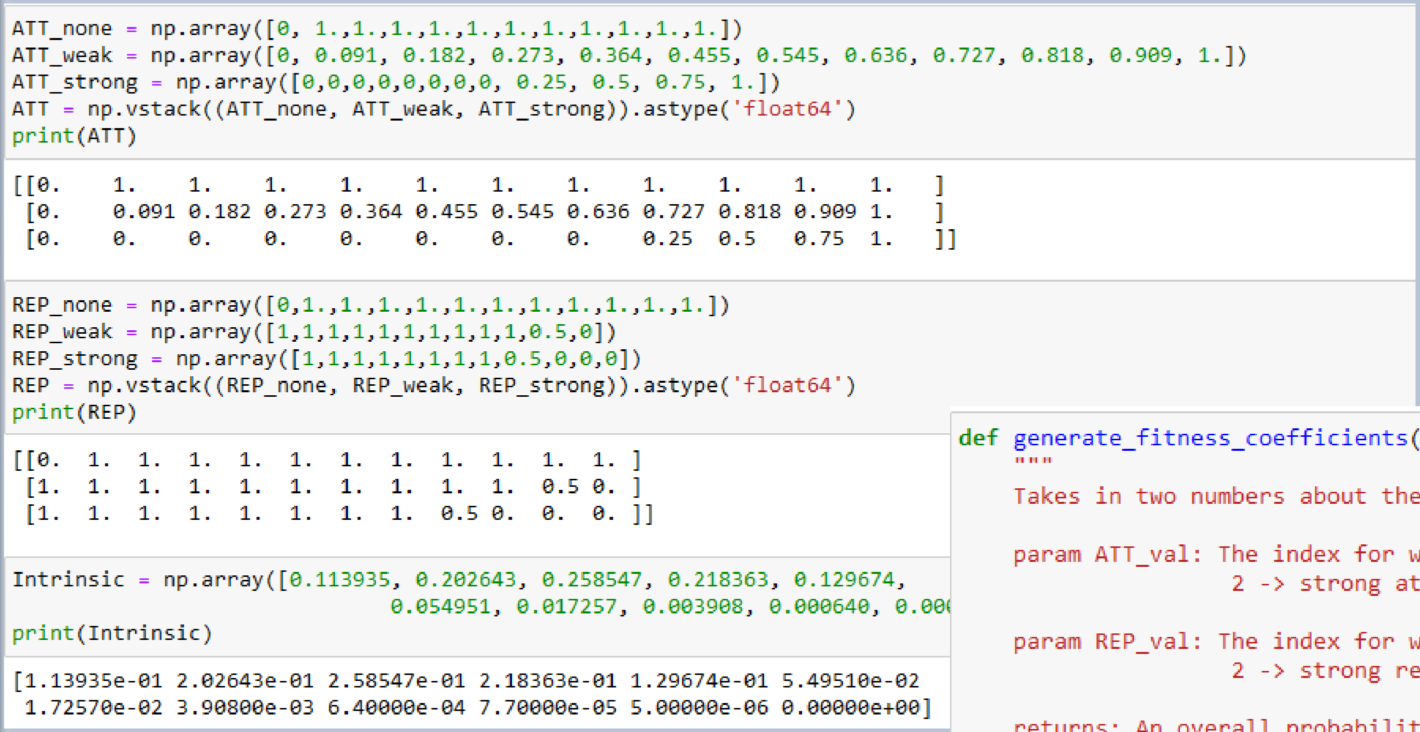
```python
def bp_distance(n1, n2):
    """
    Produces a distance measurement based off of matching

    param n1: The first nucleotide sequence

    param n2: The second nucelotide sequence

    returns: A score of how many nucleotides match

    """
    tot = 0
    for i,j in zip(n1, n2):
        if i == j:
            tot += 1
    return tot
```



```python
F = 11
consensus = produce_random_string_of_base_pairs(F)
```

Generating the Probability Distribution

- Generate the probability distribution for different types of attraction and repulsion



```
ATT_none = np.array([0, 1.,1.,1.,1.,1.,1.,1.,1.,1.,1.,1.])
ATT_weak = np.array([0, 0.091, 0.182, 0.273, 0.364, 0.455, 0.545, 0.636, 0.727, 0.818, 0.909, 1.])
ATT_strong = np.array([0,0,0,0,0,0,0,0, 0.25, 0.5, 0.75, 1.])
ATT = np.vstack((ATT_none, ATT_weak, ATT_strong)).astype('float64')
print(ATT)
```

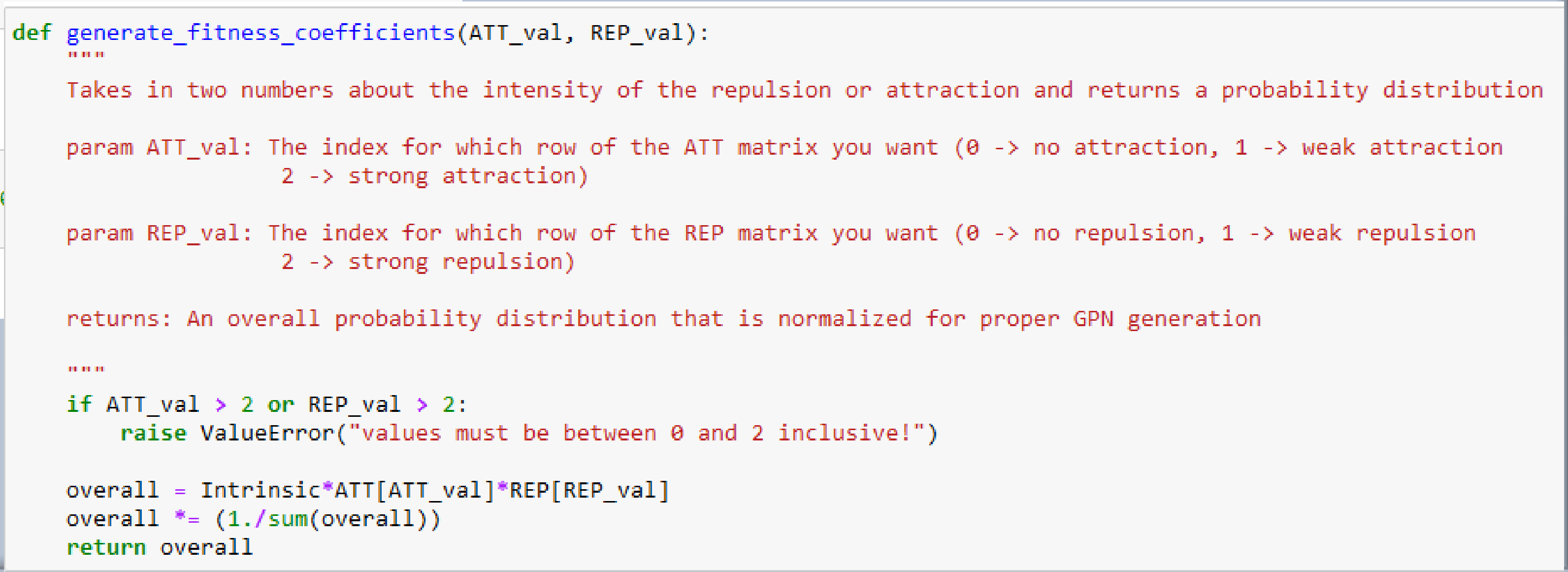
```
[[0.    1.    1.    1.    1.    1.    1.    1.    1.    1.    1.    1.   ]
 [0.    0.091 0.182 0.273 0.364 0.455 0.545 0.636 0.727 0.818 0.909 1.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.25  0.5   0.75  1.   ]]
```

```
REP_none = np.array([0,1.,1.,1.,1.,1.,1.,1.,1.,1.,1.,1.])
REP_weak = np.array([1,1,1,1,1,1,1,1,1,1,0.5,0])
REP_strong = np.array([1,1,1,1,1,1,1,1,0.5,0,0,0])
REP = np.vstack((REP_none, REP_weak, REP_strong)).astype('float64')
print(REP)
```

```
[[0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.  0.  0. ]]
```

```
Intrinsic = np.array([0.113935, 0.202643, 0.258547, 0.218363, 0.129674,
                      0.054951, 0.017257, 0.003908, 0.000640, 0.00
print(Intrinsic)
```

```
[1.13935e-01 2.02643e-01 2.58547e-01 2.18363e-01 1.29674e-01 5.49510e-02
 1.72570e-02 3.90800e-03 6.40000e-04 7.70000e-05 5.00000e-06 0.00000e+00]
```



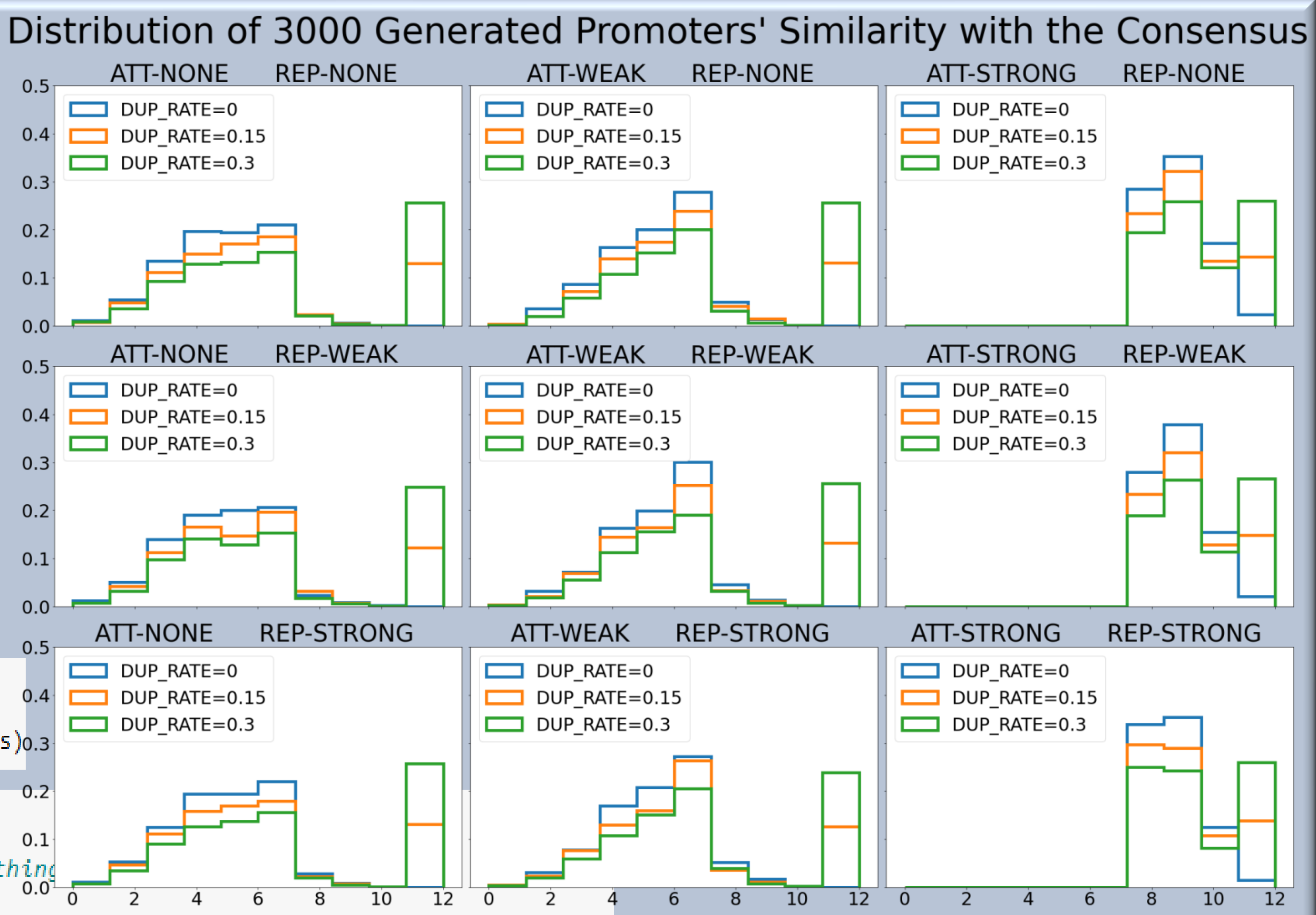
```
def generate_fitness_coefficients(ATT_val, REP_val):
    """
    Takes in two numbers about the intensity of the repulsion or attraction and returns a probability distribution

    param ATT_val: The index for which row of the ATT matrix you want (0 -> no attraction, 1 -> weak attraction
                   2 -> strong attraction)

    param REP_val: The index for which row of the REP matrix you want (0 -> no repulsion, 1 -> weak repulsion
                   2 -> strong repulsion)

    returns: An overall probability distribution that is normalized for proper GPN generation

    """
    if ATT_val > 2 or REP_val > 2:
        raise ValueError("values must be between 0 and 2 inclusive!")

    overall = Intrinsic*ATT[ATT_val]*REP[REP_val]
    overall *= (1./sum(overall))
    return overall
```

Sequences Generated Relative to the "Consensus"

- Sequences are generated by either duplicating the consensus or by sampling over the generated probability distribution for the base pairs shared.
- The consensus was then changed to match the number of base pairs that had to be shared, then inserted

```python
bp_sharing = 0
while(bp_sharing == 0): #something always needs to be shared
    bp_sharing = np.random.choice(np.arange(F+1), p=fitness_coefficients)
```

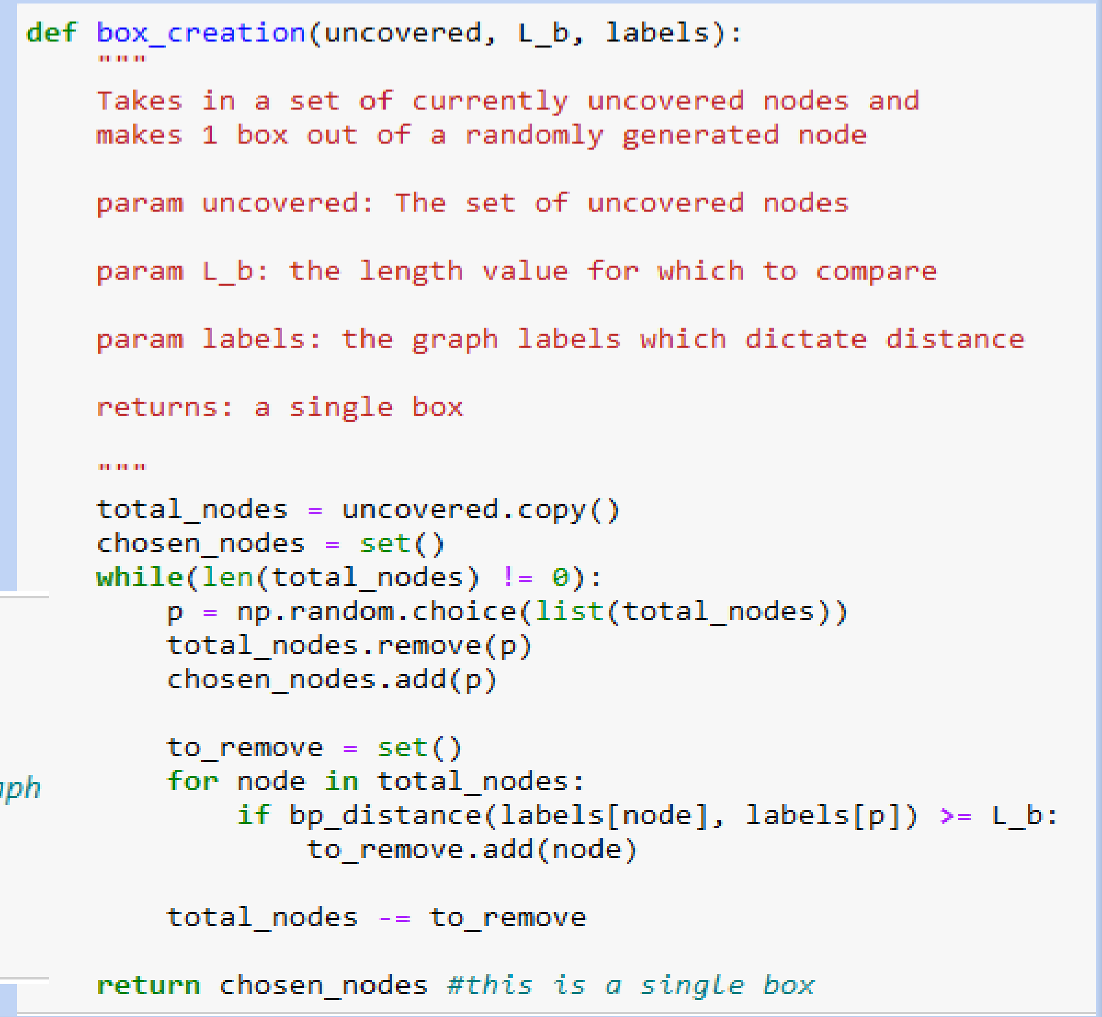
```python
for i in range(F - bp_sharing):
    index = np.random.choice(np.arange(0,F))
    while(changed.get(index) != None): #keep looping until something
        index = np.random.choice(np.arange(0,F))
    changed[index] = True
    new_promoter = new_promoter[:index] + np.random.choice(["A","T","G","C"])[0] + new_promoter[index + 1:]
```



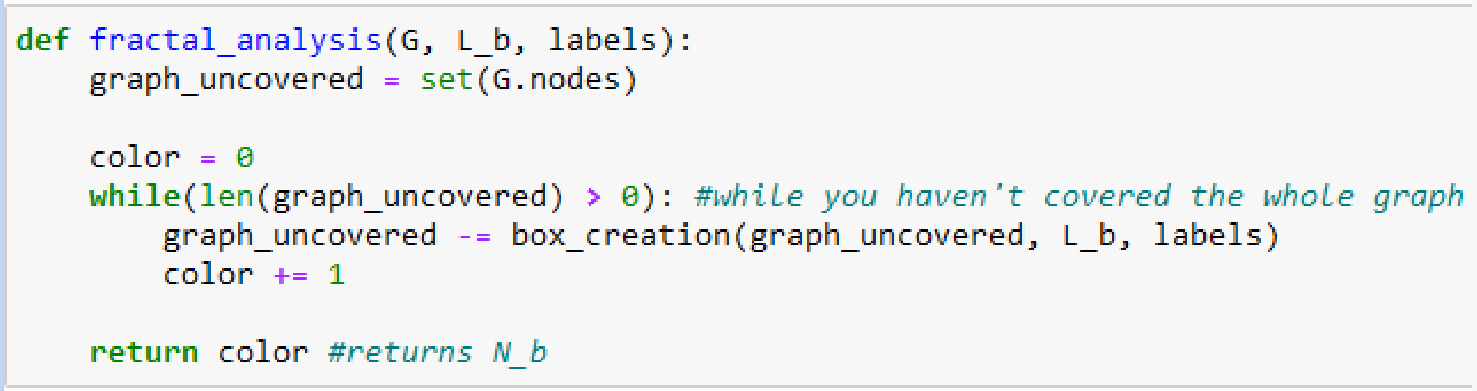
Fractally Analyzing a Generated Graph

- *Fractal_analysis* returns N_B, which is the number of boxes of distance l_B required to cover the graph given a certain chemical distance between nodes

- From this, the fractal dimension can be solved for using the relation $\frac{N_B}{N} = l_b^{-d_B}$.



```python
def box_creation(uncovered, L_b, labels):
    """
    Takes in a set of currently uncovered nodes and
    makes 1 box out of a randomly generated node

    param uncovered: The set of uncovered nodes

    param L_b: the length value for which to compare

    param labels: the graph labels which dictate distance

    returns: a single box

    """
    total_nodes = uncovered.copy()
    chosen_nodes = set()
    while(len(total_nodes) != 0):
        p = np.random.choice(list(total_nodes))
        total_nodes.remove(p)
        chosen_nodes.add(p)

        to_remove = set()
        for node in total_nodes:
            if bp_distance(labels[node], labels[p]) >= L_b:
                to_remove.add(node)

        total_nodes -= to_remove

    return chosen_nodes #this is a single box
```

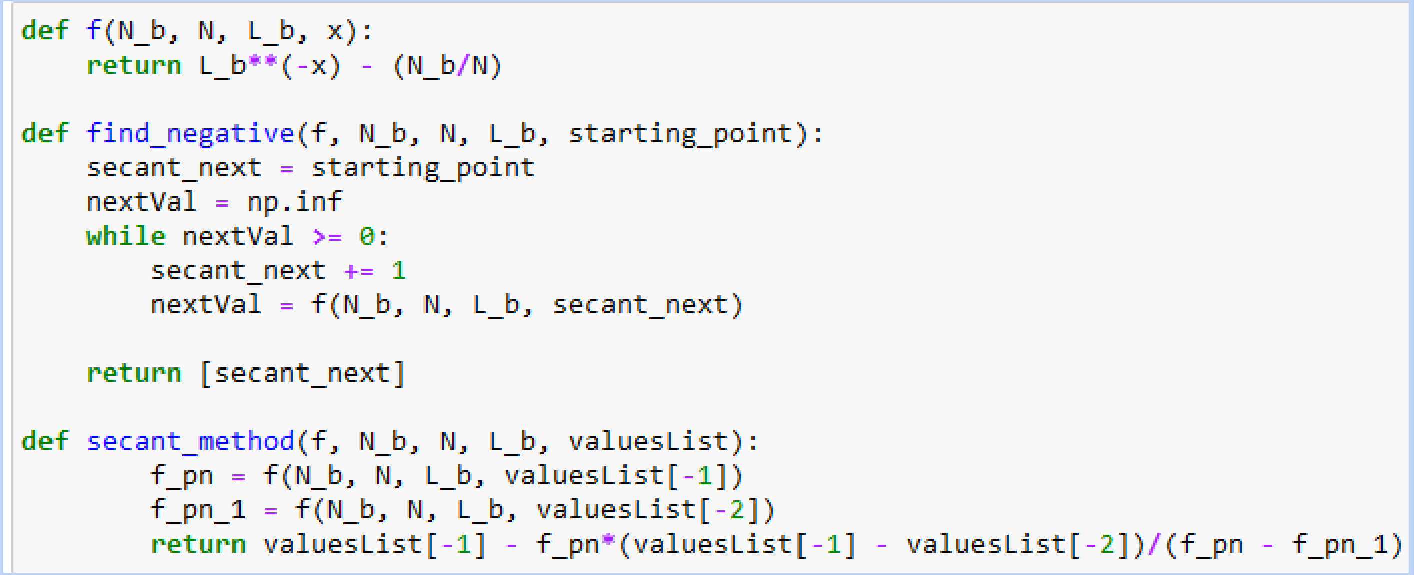


```python
def fractal_analysis(G, L_b, labels):
    graph_uncovered = set(G.nodes)

    color = 0
    while(len(graph_uncovered) > 0): #while you haven't covered the whole graph
        graph_uncovered -= box_creation(graph_uncovered, L_b, labels)
        color += 1

    return color #returns N_b
```

Fractally Analyzing a Generated Graph

- Traditional root finding methods can be used to solve $f(\mathrm{d_B}) = l_b^{-d_B} - \frac{N_B}{N} = 0$ for d_B.
- The root was found using the secant method, which requires an interval $[a, b]$ such that $f(a) > 0$ & $f(b) < 0$.
  - Such an interval is guaranteed, as $\frac{N_B}{N} \leq 1$, therefore $f(0) \geq 0$
  - As d_B increases, the function will become negative at some value $d_B{}'$

- This wasn't done in the paper, but I decided to solve the equation this way instead of an intensive regression to solve the problem



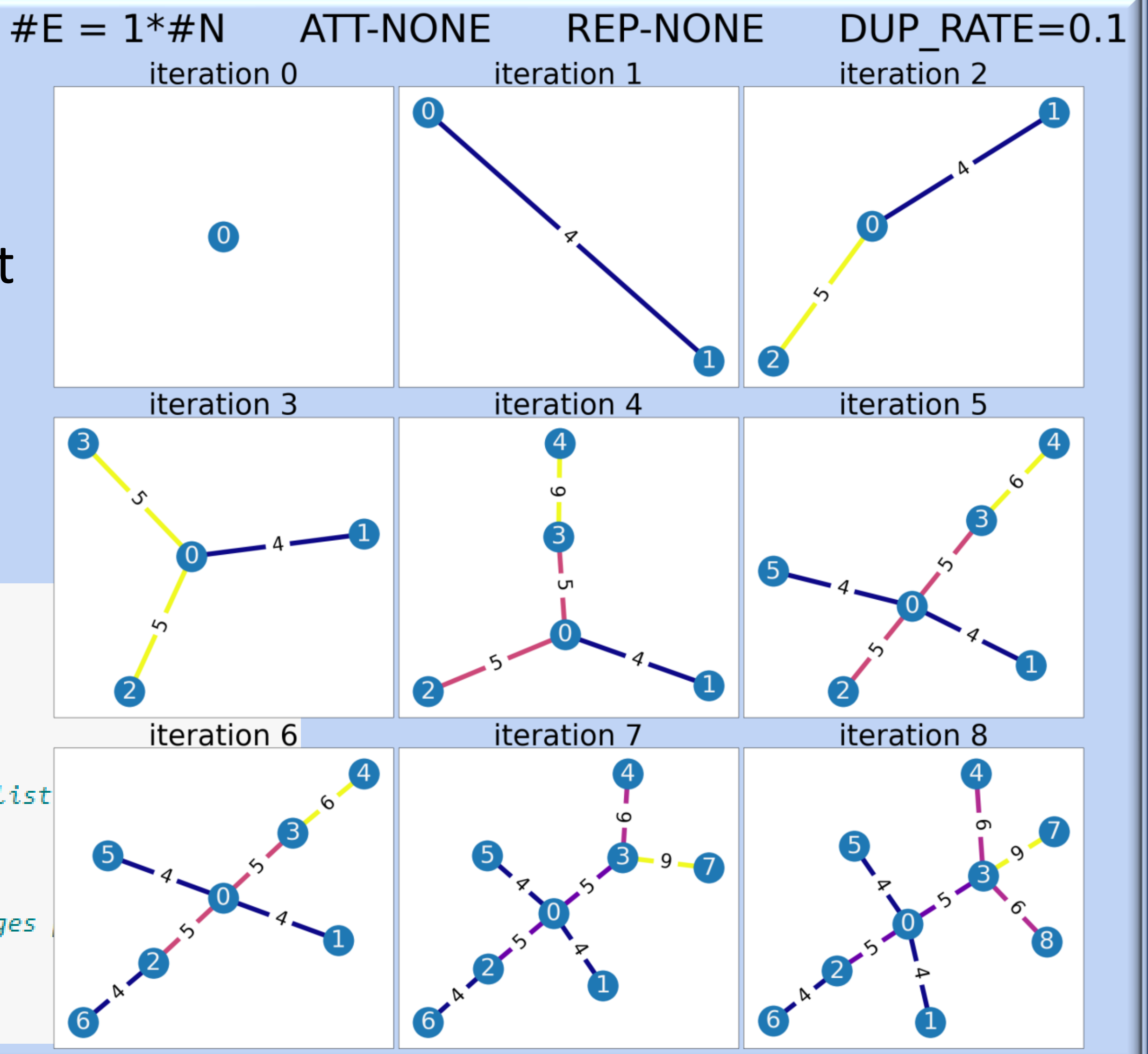
```python
def f(N_b, N, L_b, x):
    return L_b**(-x) - (N_b/N)

def find_negative(f, N_b, N, L_b, starting_point):
    secant_next = starting_point
    nextVal = np.inf
    while nextVal >= 0:
        secant_next += 1
        nextVal = f(N_b, N, L_b, secant_next)

    return [secant_next]

def secant_method(f, N_b, N, L_b, valuesList):
        f_pn = f(N_b, N, L_b, valuesList[-1])
        f_pn_1 = f(N_b, N, L_b, valuesList[-2])
        return valuesList[-1] - f_pn*(valuesList[-1] - valuesList[-2])/(f_pn - f_pn_1)
```

The Actual Algorithm (Part 1)

- First the function generates a graph with *iteration* nodes
  - Each node is connected to a certain number of others, with the best matches coming first

- The weight of the edge is the chemical distance between the two nodes

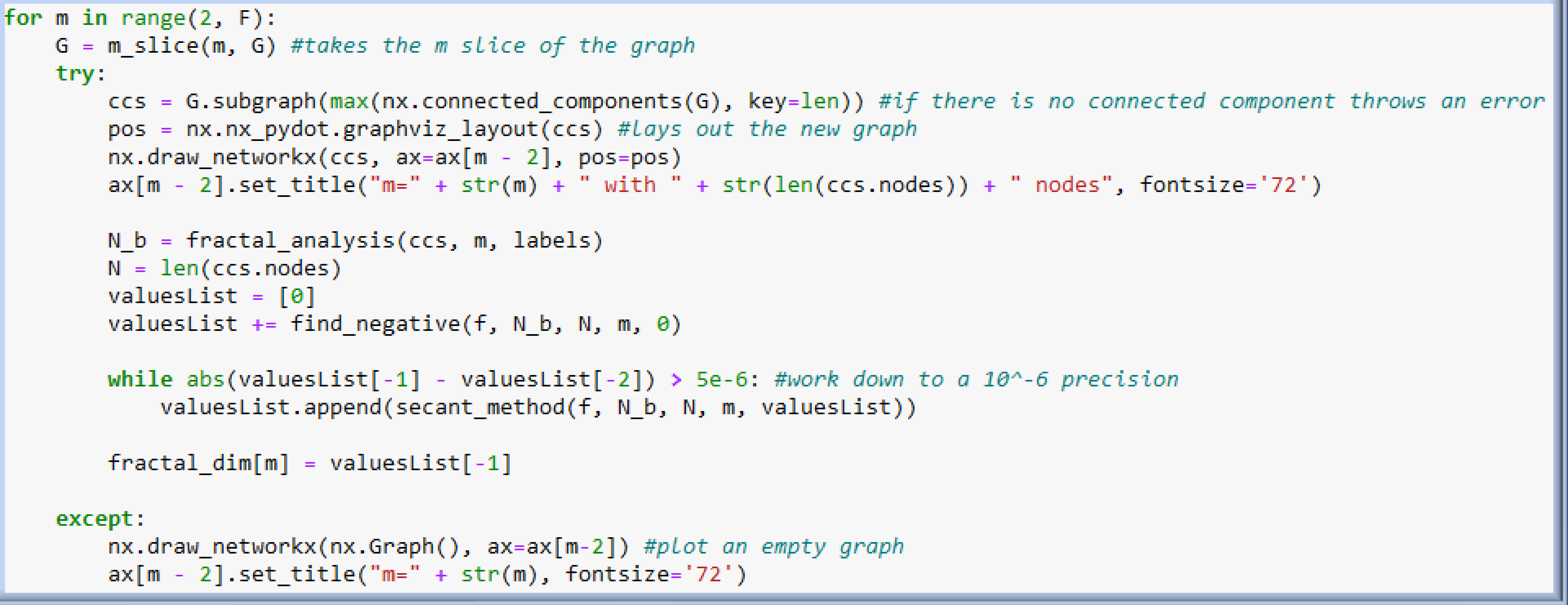
```
for i in range(iterations):
    promotion = generate_promoter(DUP_RATE, consensus, REF_PROB_LIST)
    labels[i] = promotion
    G.add_node(i)
    edges_to_add = []
    for j in range(i):
        edges_to_add.append((i,j, dist_func(labels[i], labels[j]))) #adds to the list

    if mult:
        edges_to_add = sorted(edges_to_add, key=lambda tup: tup[2], reverse=True)
        edges_to_add = edges_to_add[:mult] #only adds the multiplier number of edges
        G.add_weighted_edges_from(edges_to_add)
    else:
        G.add_weighted_edges_from(edges_to_add)
```



The Actual Algorithm (Part 2)

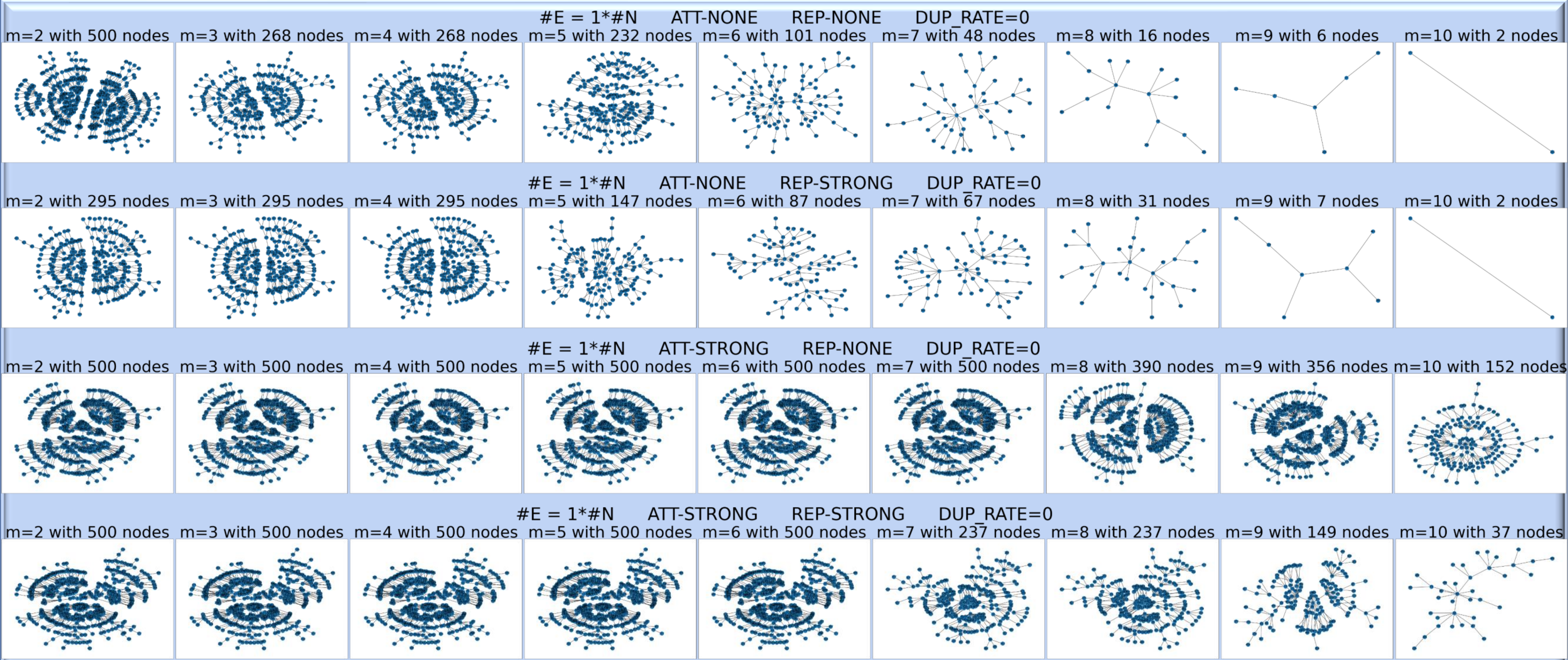
- Takes the m-slice of the graph, where all edges < m, and any node isolates as a consequence, removed

- Then the largest connected component is taken and fractally analyzed. The fractal dimension is then taken and returned for each m slice

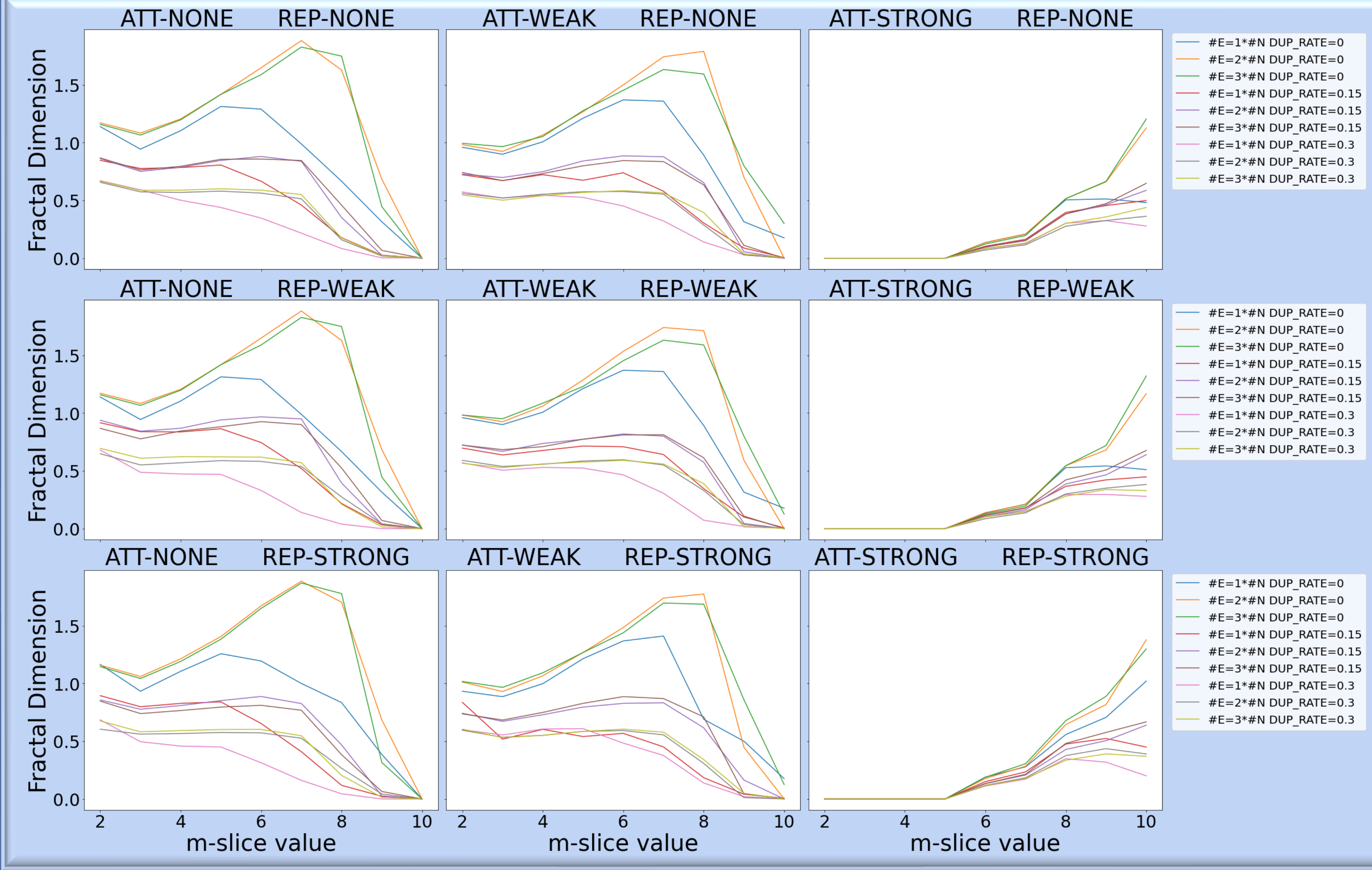


```python
for m in range(2, F):
    G = m_slice(m, G) #takes the m slice of the graph
    try:
        ccs = G.subgraph(max(nx.connected_components(G), key=len)) #if there is no connected component throws an error
        pos = nx.nx_pydot.graphviz_layout(ccs) #lays out the new graph
        nx.draw_networkx(ccs, ax=ax[m - 2], pos=pos)
        ax[m - 2].set_title("m=" + str(m) + " with " + str(len(ccs.nodes)) + " nodes", fontsize='72')

        N_b = fractal_analysis(ccs, m, labels)
        N = len(ccs.nodes)
        valuesList = [0]
        valuesList += find_negative(f, N_b, N, m, 0)

        while abs(valuesList[-1] - valuesList[-2]) > 5e-6: #work down to a 10^-6 precision
            valuesList.append(secant_method(f, N_b, N, m, valuesList))

        fractal_dim[m] = valuesList[-1]

    except:
        nx.draw_networkx(nx.Graph(), ax=ax[m-2]) #plot an empty graph
        ax[m - 2].set_title("m=" + str(m), fontsize='72')
```

Some Example Graphs



Fractal Dimension Results



Attraction seems to be a very important factor in determining the fractal status of a gene promoter network

The duplication rate seems to be the next most powerful indicator of what the fractal dimension will be