Problem 1.

(a) (Slide 11) Implement the pj|i matrix. Do yourself a favor and make it a function because you're going to use it quite a bit. You can make your function take a single shared σ 2 i = σ 2 . Sanity check: each row should sum to 1. Side note: if you decide to do the extra credit (see (k)), you should allow your algorithm to utilize different σ 2 i otherwise you're gonna have a bad time.

```python
# for part (a)
def compute_denominators_for_p(data, x_i, sigma_sq):
    x_ = data[x_i]
    res = 0

    for i in range(data.shape[0]):
        if i != x_i:
            neighbor = data[i]
            distance = np.exp(-math.dist(x_ , neighbor) ** 2 / (2 * sigma_sq))

            res += distance

    return res

def compute_p_x_on_y(data, x_i, x_j, sigma_sq):
    x_1 = data[x_i]
    x_2 = data[x_j]
    numerator = np.exp(-math.dist(x_1 , x_2) ** 2 / (2 * sigma_sq))
    denominator = compute_denominators_for_p(data, x_i, sigma_sq)

    p_ij = numerator / denominator

    return p_ij
```

```python
# part (a)
q1_matrix = np.zeros((final_data.shape[0],final_data.shape[0]))
for i in range(final_data.shape[0]):
    for j in range(final_data.shape[0]):
        if(i != j):
            q1_matrix[i][j] = compute_p_x_on_y(np.array(final_data), i, j, 0.1)

q1_matrix
```

97] ✓ 57.3s

```
array([[0.00000000e+00, 9.49713147e-02, 5.59260152e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [9.96584061e-02, 0.00000000e+00, 3.61415115e-08, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [1.33033076e-06, 8.19276655e-08, 0.00000000e+00, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       ...,
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        0.00000000e+00, 6.85430292e-30, 2.20243197e-08],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        1.29311741e-29, 0.00000000e+00, 4.23367135e-16],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        5.53093238e-09, 5.63557617e-17, 0.00000000e+00]])
```

```python
for i in range(len(q1_matrix)):
    print(' index',i, 'the sume is:',q1_matrix[i].sum())
```

98] ✓ 0.4s

```
index 0 the sume is: 0.9999999999999999
index 1 the sume is: 0.9999999999999999
index 2 the sume is: 1.0
index 3 the sume is: 0.9999999999999998
index 4 the sume is: 0.9999999999999999
index 5 the sume is: 1.0000000000000002
index 6 the sume is: 0.9999999999999999
index 7 the sume is: 0.9999999999999999
index 8 the sume is: 1.0
index 9 the sume is: 1.0
index 10 the sume is: 0.9999999999999999
index 11 the sume is: 1.0000000000000002
index 12 the sume is: 1.0000000000000004
index 13 the sume is: 1.0
index 14 the sume is: 1.0
index 15 the sume is: 0.9999999999999999
```

(b) (Slide 11) Implement the pij matrix. Sanity check: the entire matrix should sum to 1

```python
# for part (b)
def compute_p_ij(data, sigma_sq):
    num_pts = data.shape[0]
    matrix = np.zeros((num_pts, num_pts))
    for i in range(num_pts):
        for j in range(num_pts):
            if i != j:
                p_i_on_j = compute_p_x_on_y(np.array(data), i, j, sigma_sq)
                p_j_on_i = compute_p_x_on_y(np.array(data), j, i, sigma_sq)
                p_ij =  (p_j_on_i + p_i_on_j) / (2 * num_pts)
                matrix[i][j] = p_ij

    return matrix
```

(b). Implement the pij matrix

```
# part (b)

q2_matrix = compute_p_ij(final_data, 0.1)
q2_matrix
```

9] ✓ 1m 22.9s

```
array([[0.00000000e+00, 4.86574302e-04, 4.72397727e-09, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [4.86574302e-04, 0.00000000e+00, 2.95172943e-10, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [4.72397727e-09, 2.95172943e-10, 0.00000000e+00, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       ...,
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        0.00000000e+00, 4.94636926e-32, 6.88881301e-11],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        4.94636926e-32, 0.00000000e+00, 1.19930724e-18],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        6.88881301e-11, 1.19930724e-18, 0.00000000e+00]])
```

```
q2_matrix.sum()
```

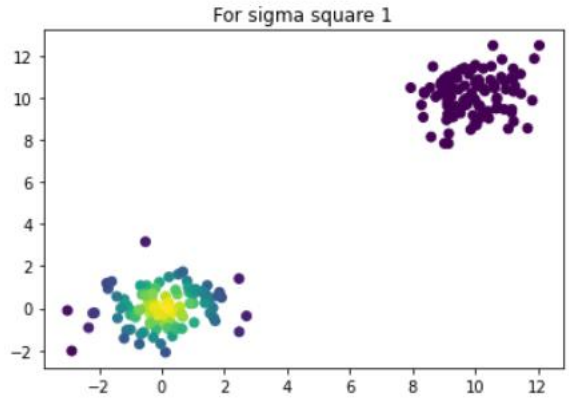
0] ✓ 0.2s

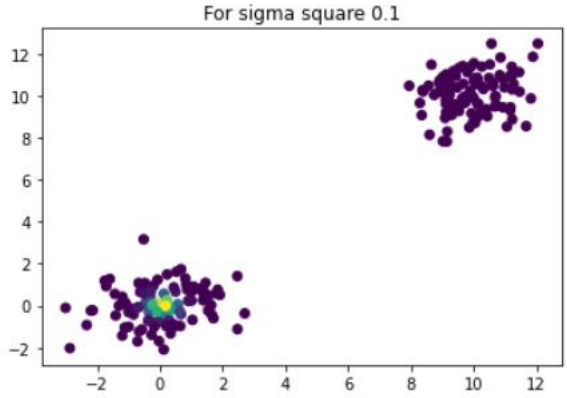
```
1.0
```

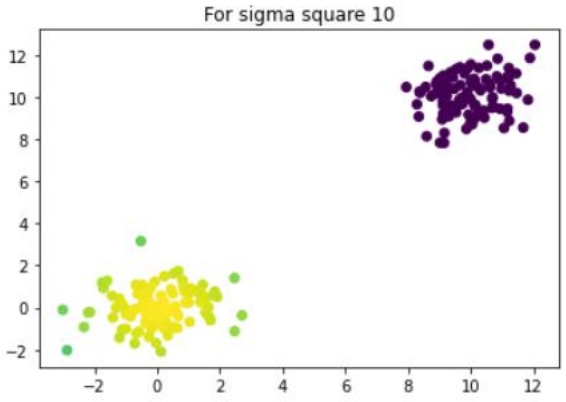
(c) Using σ 2 = 1, plot the entire dataset and color the points based on their probability relative to the first data point. To be rigorous: p1j is the vector of probabilities of "j picking 1 as its neighbor". A reasonable color scale might be: wj $\propto$ p1j/maxk(p1k). Your plot should show a change of color away from the first data point. Do the same for σ 2 = {0.1, 10, 100}. Each plot might look something like this:

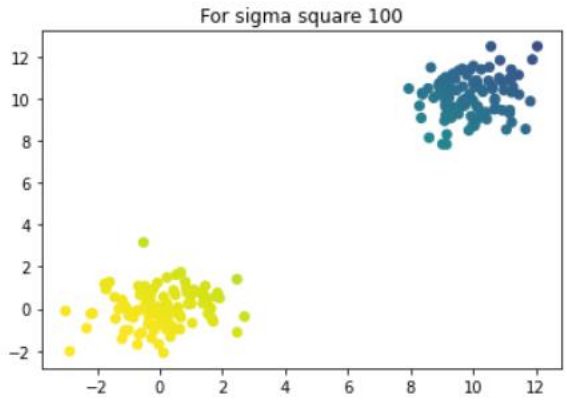
```
# part (c)
q3_matrix_1 = q2_matrix
q3_matrix_2 = compute_p_ij(final_data, 1)
q3_matrix_3 = compute_p_ij(final_data, 10)
q3_matrix_4 = compute_p_ij(final_data, 100)
```

01] ✓ 3m 52.7s


For sigma square 1
12
10
8
6
4
2
0
−2
−2
0
2
4
6
8
10
12


For sigma square 0.1
12
10
8
6
4
2
0
−2
−2
0
2
4
6
8
10
12


For sigma square 10
12
10
8
6
4
2
0
−2
−2
0
2
4
6
8
10
12



(d) (Slide 13) Implement the qij matrix. Sanity check: the entire matrix should sum to 1.

(d) Implement the qij matrix. Sanity check: the entire matrix should sum to 1.

```
def compute_denominator_for_q(data, y_i):
    y_ = data[y_i]
    res = 0

    for i in range(data.shape[0]):
        if i != y_i:
            neighbor = data[i]
            distance = (1+math.dist(y_,neighbor)**2)**(-1)

            res += distance

    return res

def compute_q_x_on_y(data, y_i, y_j):
    y_1 = data[y_i]
    y_2 = data[y_j]
    numerator = (1+math.dist(y_1,y_2)**2)**(-1)
    denominator = compute_denominator_for_q(data, y_i)

    q_ij = numerator / denominator

    return q_ij


def compute_q_ij(data):
    num_pts = data.shape[0]
    matrix = np.zeros((num_pts, num_pts))

    for i in range(num_pts):
        for j in range(num_pts):
            if i != j:
                q_ij = compute_q_x_on_y(np.array(data), i, j)
                matrix[i][j] = q_ij / num_pts



    return matrix
```

✓ 3.2s

```
q_ij_matrix = compute_q_ij(np.array(final_data))
```
[226] ✓ 39.9s

```
q_ij_matrix.sum()
```
[280] ✓ 0.4s

... 1.0

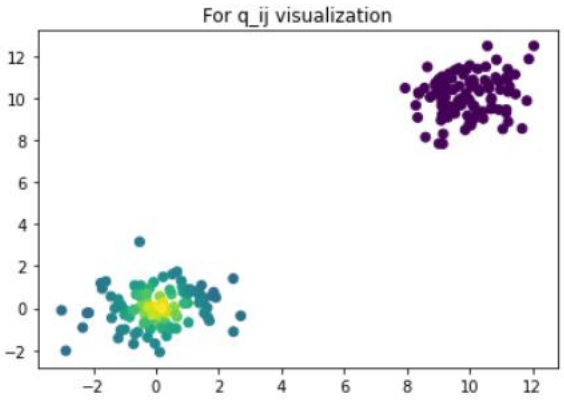
(e) Plot the entire dataset and color the points based on their q1j probability relative to the first data point. How is it different than the p1j plot from (c) when 2 i = 1?

(e) Plot the entire dataset and color the points

```
p_1j = q_ij_matrix[:,0]

plt.title('For q_ij visualization')
plt.scatter(final_data[0], final_data[1], c=p_1j)
```
28] ✓ 1.3s

.. <matplotlib.collections.PathCollection at 0x11d2b7fa0>



It looks pretty similar to sigma square equal to 1 for the p1j plot, except that for q_ij the first clusters' color is lighter than that of the p_1j

(f) (Slide 13) Implement the KL-divergence. Note, the contribution of {ij} is zero when pij = 0.

(f). Implement the KL-divergence. Note, the contribution of {ij} is zero when pij = 0.

```
def kl_divergence(p_matrix, q_matrix):
    sum = 0
    for i in range(p_matrix.shape[0]):
        for j in range(q_matrix.shape[0]):
            if p_matrix[i][j] != 0: #if p is 0, then there's no contribution of {ij} we can ignore it
                sum = sum + p_matrix[i][j] * np.log(p_matrix[i][j] / q_matrix[i][j])

    return sum
```

✓ 0.3s

(g) Using the real data as the low-dimensional projection, compute the KL-divergence when: i. 2 = 0.1. ii. 2 = 1. iii. 2 = 100. Any thoughts on what might be happening?

```
q_matrix = q_ij_matrix
p_matrix_1 = compute_p_ij(final_data, 0.1)#here the sigma i tested in q2 is 0.1
p_matrix_2 = compute_p_ij(final_data, 1) #where the sigma is 1
p_matrix_3 = compute_p_ij(final_data, 100) #where the sigma is 100
```

✓ 6m 1.1s

```
kl_divergence(p_matrix_1, q_matrix)
```

✓ 0.4s

```
1.278147252547255
```

```
kl_divergence(p_matrix_2, q_matrix)
```

✓ 1.2s

```
0.10326654539714238
```

```
kl_divergence(p_matrix_3, q_matrix)
```

✓ 1.6s

```
0.7067671570631093
```

The values of sigma affects p_ij. As we can see from the graphs that q_ij looks pretty similar to the p_ij with sigma equal to 1. KL divergence is a measurement of how different the two distributions are, therefore, we get a lower KL score when sigma is equal to 1.

(h) Using σ 2 = 1, can you find a projection that reduces the KL-divergence? Note, there are plenty linear or non-linear ones. The easiest might be to do might be 'move' one cluster. Plot the projection and report the KL-divergence.

```
temp = final_data[100:200] +5
temp1 = final_data[0:100]

projection = temp1.append(temp)
projection
```

✓ 0.1s

	0	1
0	0.072435	0.004788
1	-0.093958	0.118346
2	1.637295	0.021868
3	0.266178	-0.042178
4	0.672105	-0.403411
...	...	...
195	14.864852	13.486013
196	14.934785	14.162691
197	16.232787	13.870784
198	12.948785	15.471377
199	15.613488	15.619704

200 rows × 2 columns

```
q_matrix_projection = compute_q_ij(projection)
p_matrix_projection = compute_p_ij(projection, 1) #where the sigma is 1
```

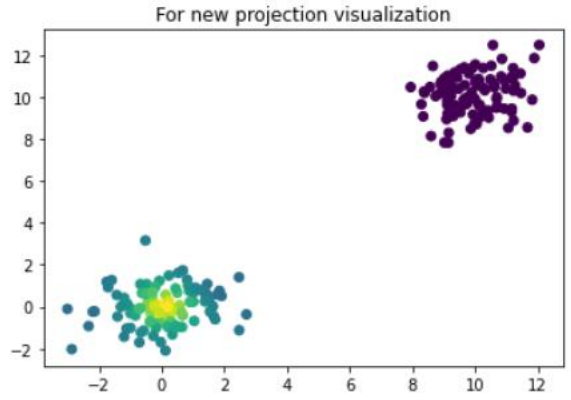
✓ 1m 51.9s

```
q = q_matrix_projection[:,0]

plt.title('For new projection visualization')
plt.scatter(final_data[0], final_data[1], c=q)
```

✓ 2.2s

```
<matplotlib.collections.PathCollection at 0x11fda4b50>
```




```
kl_divergence(p_matrix_projection, q_matrix_projection)
```

✓ 0.3s

```
0.09453014941412981
```

I moved the second cluster further and observed a smaller KL divergence value

(i) Take the first two principal components and use them as your projection. How does the KL-divergence change from when you used the real data and why? No, you don't need to implement PCA.

The KL-divergence value did not change when I used PCA on it. I think that might be due to the fact that the original data is already in 2 dimensions, and taking first and second components of PCA is equivalent to just take these 2 dimensions.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
df = pca.fit_transform(final_data)
```

✓ 0.3s

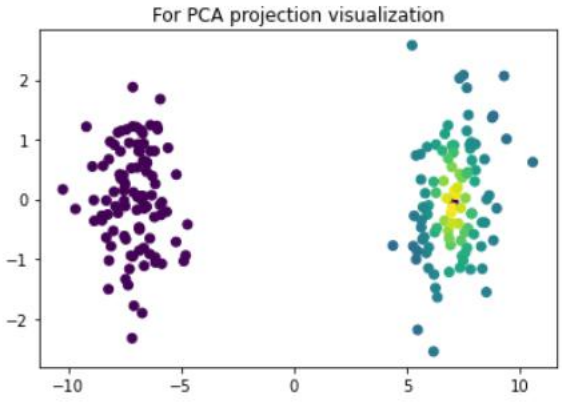
```
q_matrix_pca = compute_q_ij(df)
p_matrix_pca = compute_p_ij(df, 1) #where the sigma is 1
```

✓ 2m 10.4s

```
kl_divergence(p_matrix_pca, q_matrix_pca)
```

✓ 0.3s

```
0.10326654539714228
```



(j) Summarize your thoughts on how these hyper-parameters matter.

The sigma is the hyperparameter used in p matrix, when the hyperparameter sigma increases, the probability of the point chooses the other point as its neighbor also increases. As we can observe from the graphes plotted above for different sigmas, more points are being classified as neighbors for the first point as we increase the sigma value.

Problem 2.

Given string: banamabananas

(a). Construct the cyclic rotations.

Add a $ to the end, banamabananas$

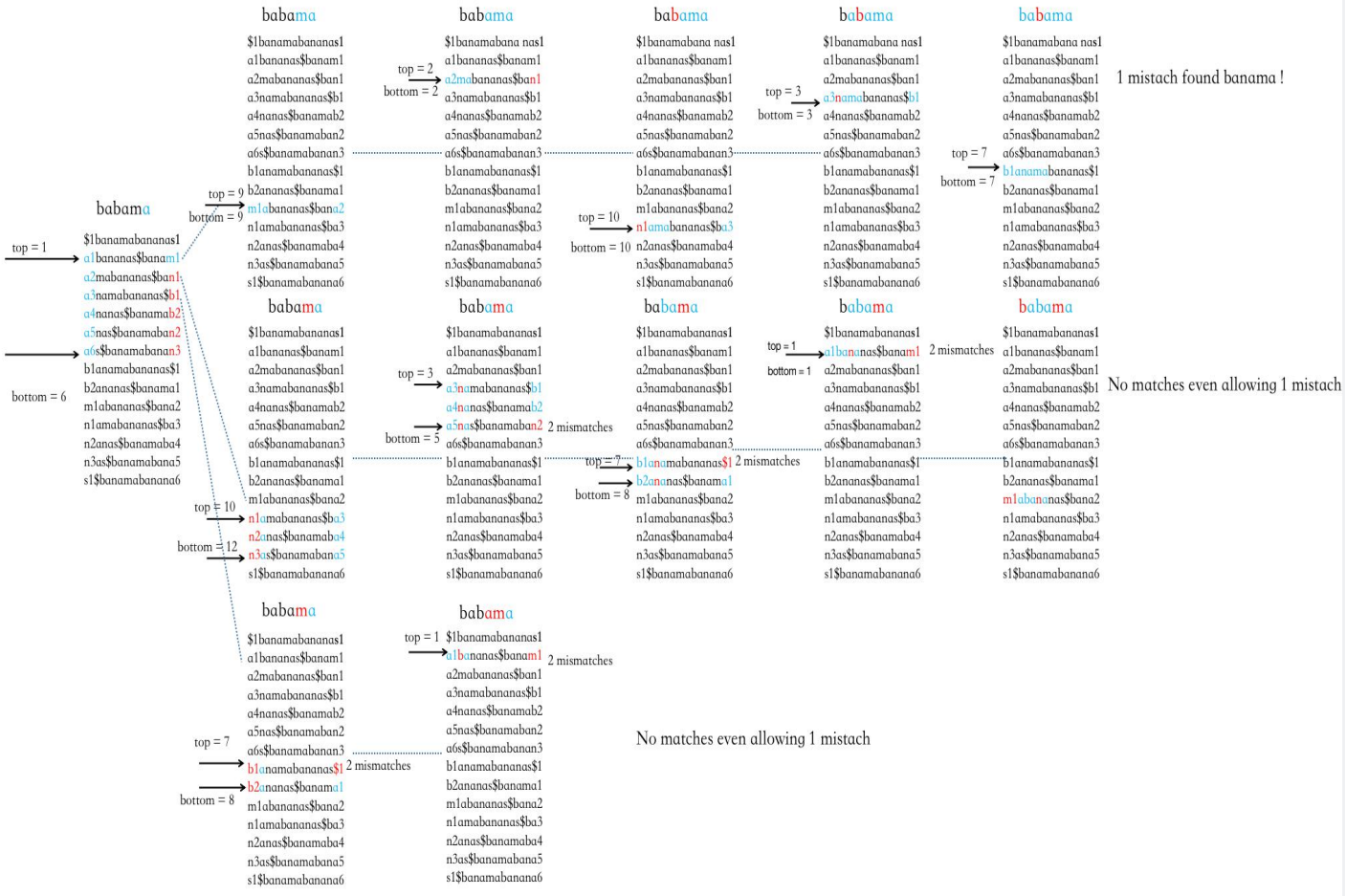
$banamabananas
s$banamabanana
as$banamabanan
nas$banamabana
anas$banamaban
nanas$banamaba
ananas$banamab
bananas$banama
abananas$banam
mabananas$bana
amabananas$ban
namabananas$ba
anamabananas$b
banamabananas$

(b). Construct the M(Text) matrix and report the BWT.

$banamabanana**s**
abananas$bana**m**
amabananas$ba**n**
anamabananas$**b**
ananas$banama**b**
anas$banamaba**n**
as$banamabana**n**
banamabananas**$**
bananas$banam**a**
mabananas$ban**a**
namabananas$b**a**
nanas$banamab**a**
nas$banamaban**a**
s$banamabanan**a**

BWT(Text) = **smnbbnn$aaaaaa**

(c). Align babama allowing one mismatch. Show the steps like in the book (Figure 9.19).

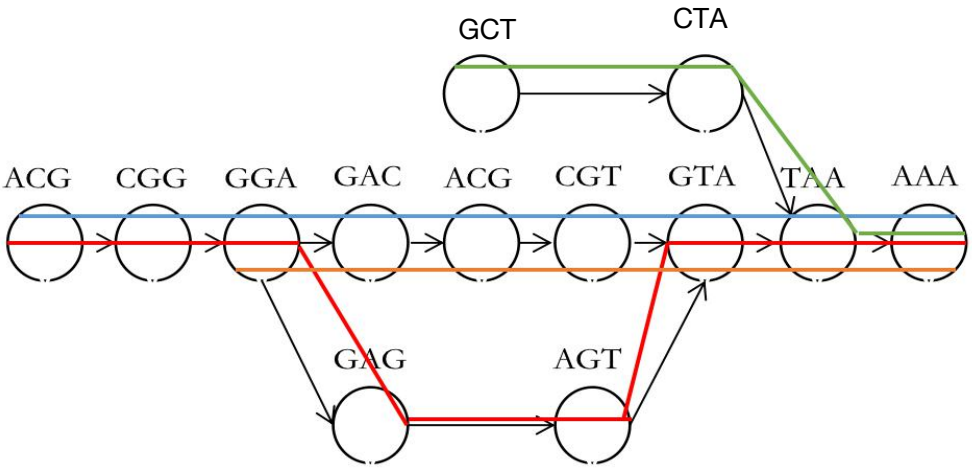


Problem 3

Pseudoalignment

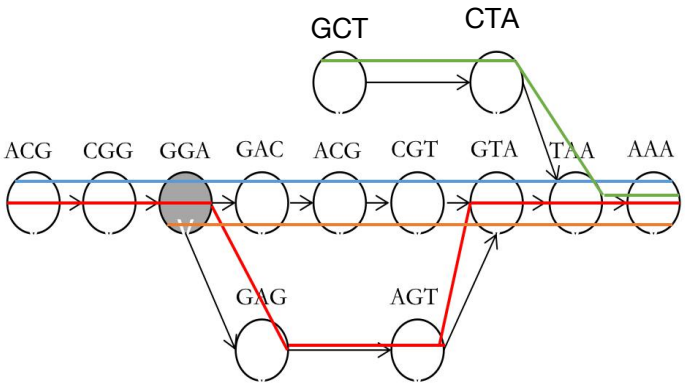
>t1_g1 ACGGACGTAAA
>t2_g1 ACGGAGTAA
>t3_g1 GGACGTAAA
>t4_g2 GCTAAA

(a) Draw the colored de Bruijn graph that one would use for pseudoalignment with k = 3. In this particular case, the nodes have 3-mers, not the edges (this is the setup we used in class). Refer to the slides on construction.

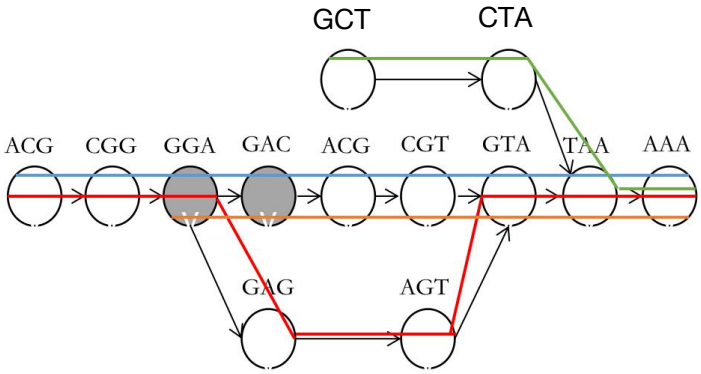


(b). Pseudoalign GGACGT and show the relevant steps you might take (including skips). Refer to the slides for the pseudoalignment.

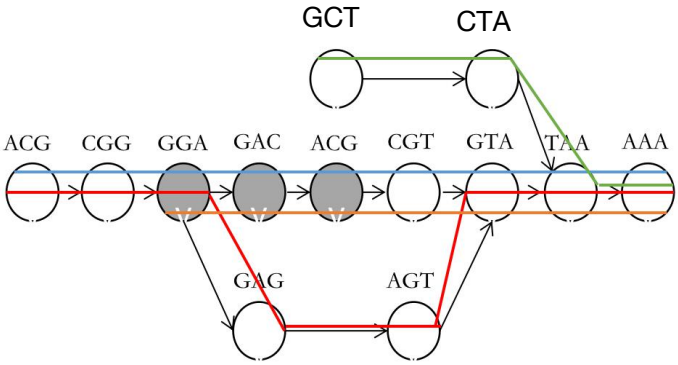
Kmers: GGA, GAC, ACG, CGT
GGA



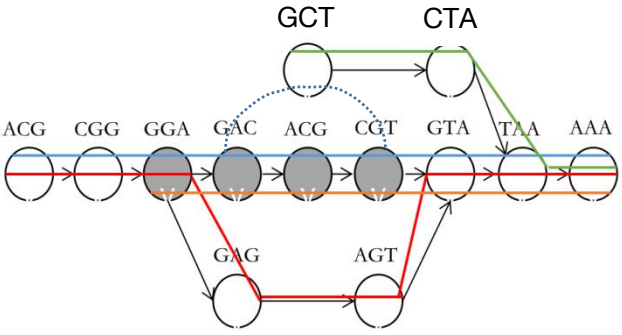
GAC



ACG



CGT

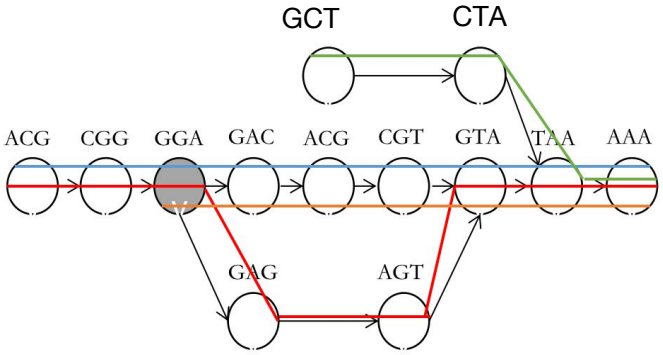


Could be a jump/skip here from GAC to CGT because of the same equivalent calsses

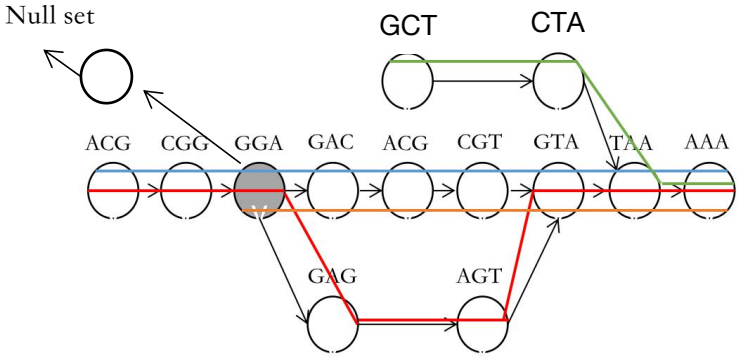
(c). **Pseudoalign GGATGT and show the relevant steps you might take (including skips). Remember, every k-mer has an equivalence class which is a set. If a k-mer is in your data but not in your transcriptome, its equivalence class is the null set.**

K-mers: GGA GAT ATG TGT

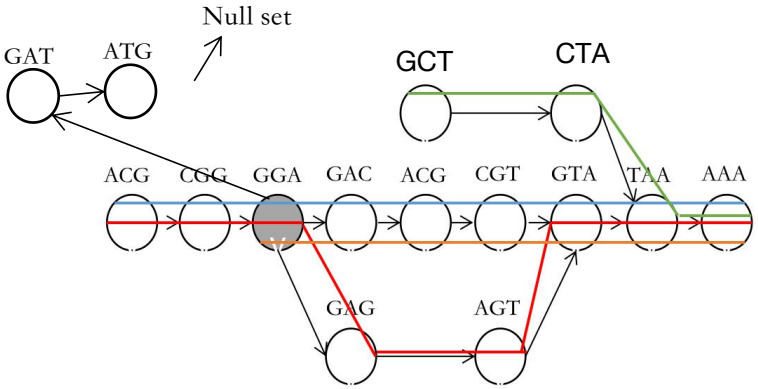
GGA



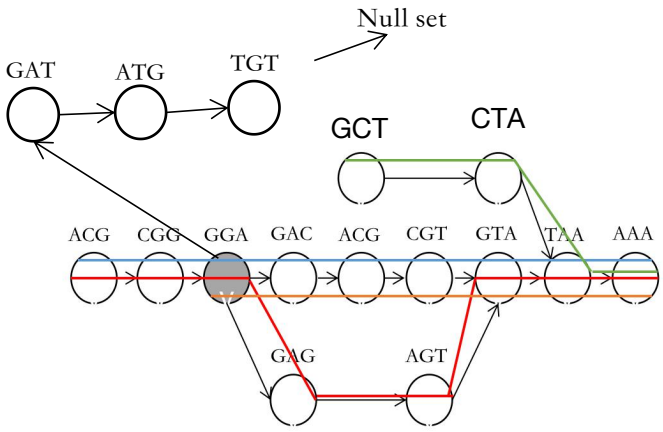
GAT's equivalent class is null set



ATG's equivalent class is null set



TGT's equivalent class is null set



(d). **The previous read might arise if there is an error at position 4 (using 1-based indexing). Describe an algorithm to deal with the error. Describe the benefits and drawbacks of your algorithm. These properties might come in speed, loss of data, or false alignments (or other things). Note: there isn't one correct answer.**

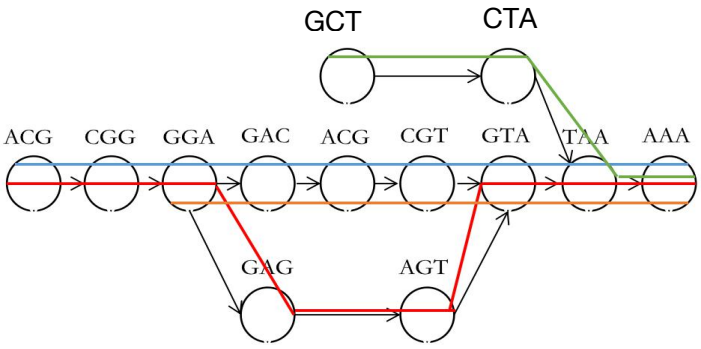
From my group, we thought of a way to deal with this kind of error might be to check k-mers on the error site with every other combinations of possibile base. For example, since we have an error at position 4 for part c. Starting from k-mer GAT we can't align it with the original graph we got. So we kind of check GAA, GAC, and GAG to see if any of these matches with our original. If it matches, then we add another node of this k-mer to the original graph. The benefit of this will be that we are able to pseudo align the k-mers with one error at any position, but the drawbacks would be the speed and also the memories needed to store these k-mers.

(e) The following sequence is a valid RNA-seq read TTTACG. Clearly it won't give you a non-empty equivalence class as-is. How did this data arise and how might you pseudoalign it? Hint: think about how RNA-seq is generated. That is, the orientation of the reads matters and can generate some annoying things we have to keep track of.

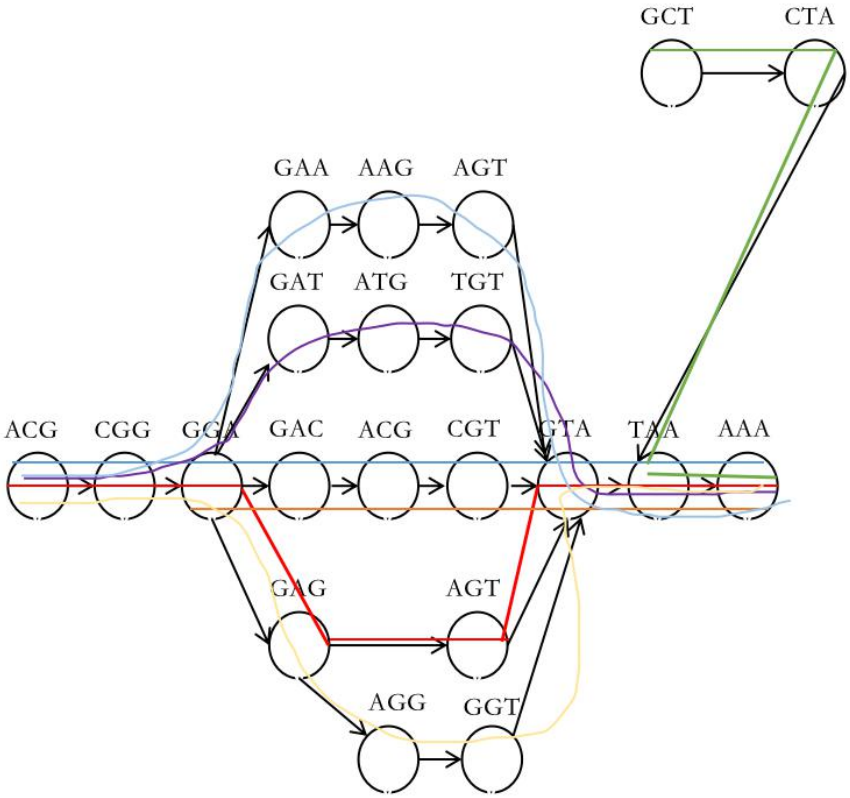
Do the complement of the reversed sequence TTTACG, which will be CGTAAA. As we can see from the de Bruijn graph we have the k-mer CGT and AAA which we can pseudoalign the sequence.

(f) Imagine you knew the haplotypes of this person and instead of an error coming from position 4, there is a single nucleotide polymorphism (SNP). Basically, one base is altered on one strand of DNA and on the other strand of DNA it is as the original reference. This means that in practice you will see reads that look like (b) and (c) in approximately equal proportion (ehh, there are caveats, but let's pretend). How might you change your colored de Bruijn graph to deal with this case?

We can just make branches out of the k-mer containing this error at position 4 for all possible bases. For example, consider the graphs below.



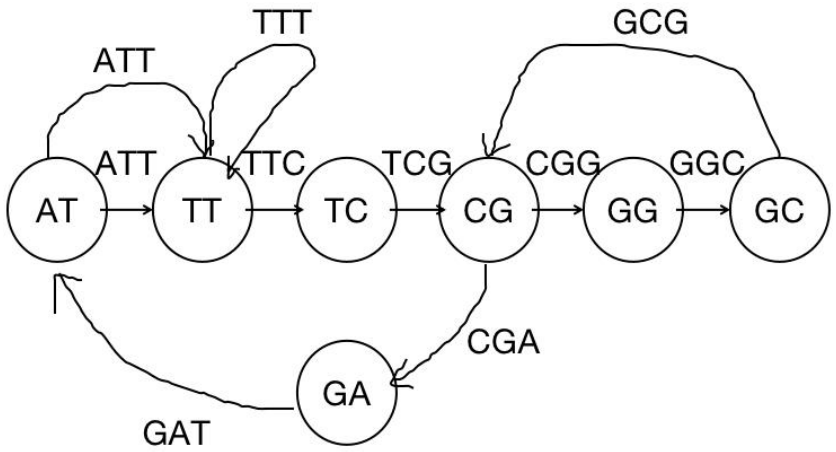
We know that the error position is at 4 for every equal proportions of the gene. Since the graph without the SNP cannot align GAT ATG and TGT, for instance, we can branch out with the possible bases that contains the error or SNP.



Problem 4

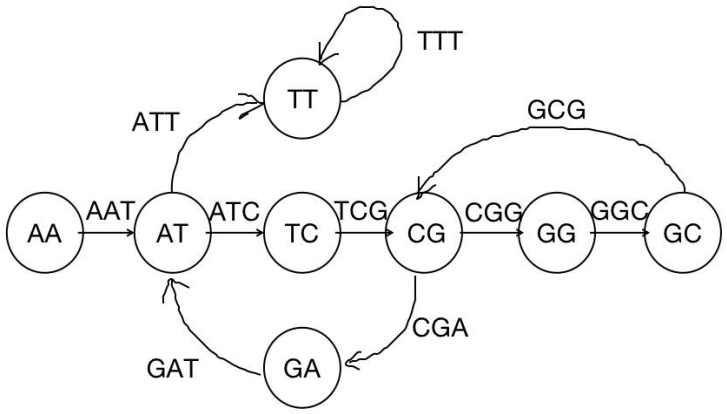
(a) Consider the following sequence: ATTCGGCGAT T T Draw the de Bruijn graph using k = 3

K-mers ATT TTC TCG CGG GGC GCG CGA GAT ATT TTT

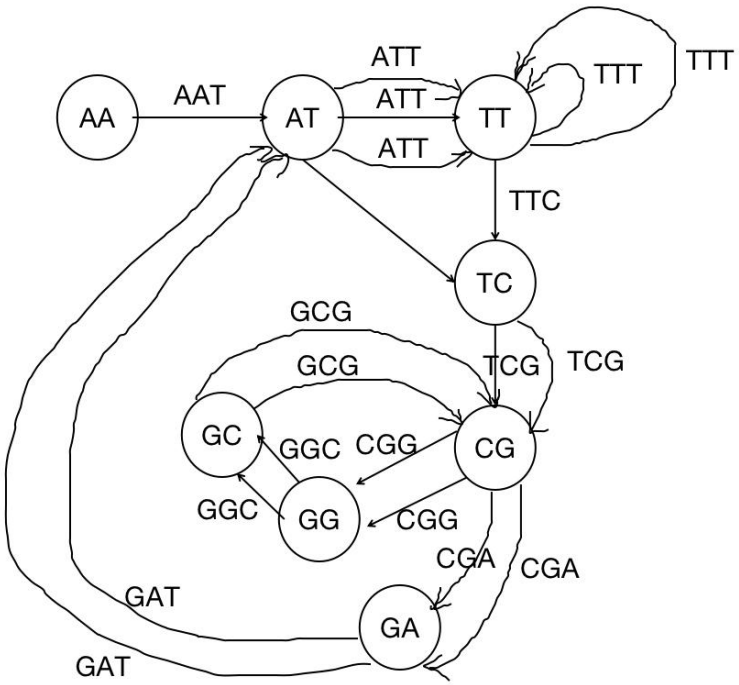


(b) We have previously only built graphs assuming a perfect sampling of the composition. Now, build the graph from the sequence: AAT CGGCGAT TT and merge it with the graph from (a). You can build two different graphs and merge them, or build the graph using the merged composition from both genomes.

K-mers: AAT ATC TCG CGG GGC GCG CGA GAT ATT TTT



Combined graph is:



(c) Is there an Eulerian path?

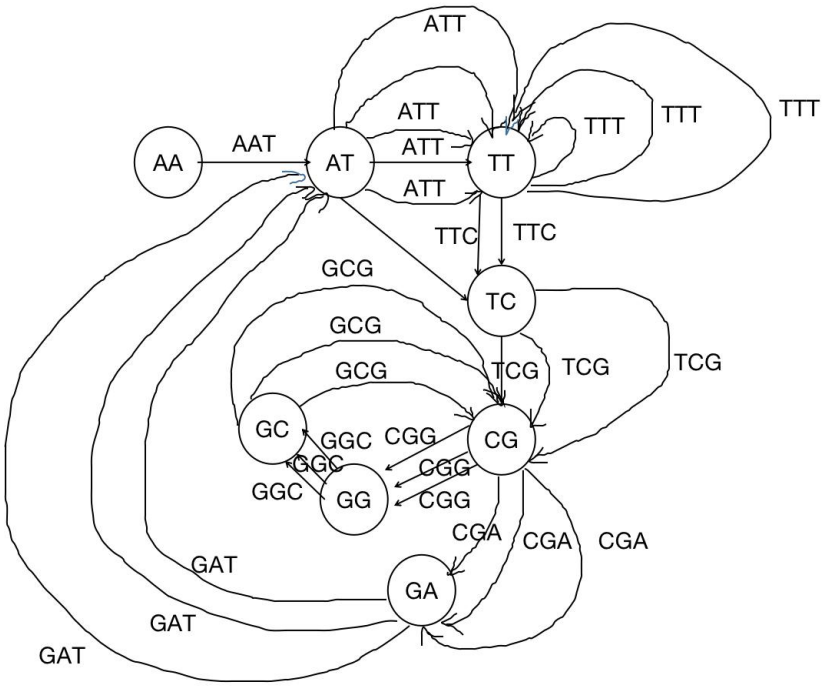
No, since there are 3 vertices with odd degrees, which is more than 2, so there is no Eulerian path

(d) Is the original string a subgraph of the graph in (b)?

Yes. The definition of a subgraph is that the nodes and edges of a graph are a subset of another graph. Since we compute the combined graph on the original string, it is a subgraph of (b).

(e) Let's build one more annoying graph. Add another instantiation of the graph from (a) to the graph from (b). Sanity check: there should be three edges going from CG to GA. Did this help our cause of reconstructing the original sequence?

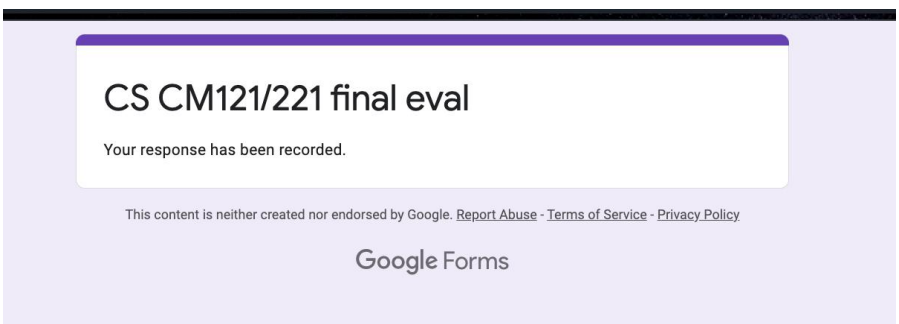
The new graph has less than 2 vertices with odd degrees, it indicates that there's a Eulerian path, so we could possibly reconstruct the original sequence.



(f) What makes assembly hard if you begin considering errors in your sampling?

We can see from the graphes in part a and b that these two sequence only differs in one base letter, they resulted in two different de Bruijn graph, one with more nodes than the other. If we were to consider errors in our sampling, we need to add more nodes to the graphes which is troublesome.

Problem 5



My **collaborators**: Jin Kim, Chris Apton