Chapter 3 :

How do we assemble genomes ?

Bioinformatics Algorithms,
Phillip Compeau & Pavel Pevzner

MODU PLS1 - Python for life science 1

Lauriane GRASS
Valentin GOUPILLE
Abdelhakim BOUAZZAOUI

M2 Bioinformatics

Introduction

Problème : Assemblage de novo, des reads au génome

Contexte :

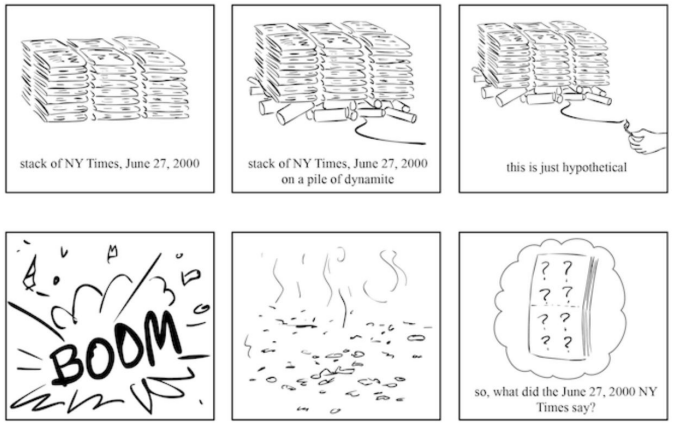
- Séquençage de novo : lecture d'un génome (inconnu)
- Technologies actuelles : lectures courtes (reads) fragmentées
- Défi : reconstituer la séquence originale complète

Le défi de l'assemblage :

- Fragments de taille limitée (150 bp)
- Chevauchements multiples à identifier

But : obtenir une chaîne génome telle que

- Tous les reads doivent faire partie de cette chaîne
- Ce génome doit être le plus “petit possible”

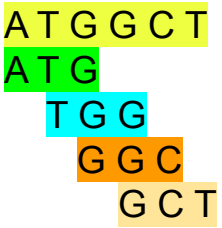


Introduction

Approche par k-mers

Les k-mers, c'est quoi ?

- Découpage des reads en morceaux de taille fixe k
- Exemple : pour k=3
  - Read "ATGGCT" → k-mers : "ATG", "TGG", "GGC", "GCT"



Pourquoi utiliser des k-mers ?

- Tous les fragments font la même taille
- Plus facile d'identifier les connexions
- Très adaptés pour construire un graphe !

Le choix de k est important :

- k petit = trop de connexions possibles
- k grand = connexions manquantes

Script 1

```python
def Compositionk(Text, k):
    """Generate the k-mer composition of a string.
    Text: A string.
    k: An integer."""
    kmers = []  # create an empty list to store the k-mers
    for i in range(len(Text) - k + 1):  # iterate over the string
        # +1 is added to len(Text) - k to include the last k-mer in the string
        kmers.append(Text[i : i + k])  # add the k-mer to the list
    # return (kmers)
    return "\n".join((kmers))
```

```python
# Test the function with the sample dataset
Text = "CAATCCAAC"
k = 5
print(Compositionk(Text, k))
```

✓ 0.0s

```
CAATC
AATCC
ATCCA
TCCAA
CCAAC
```

Script 2 *

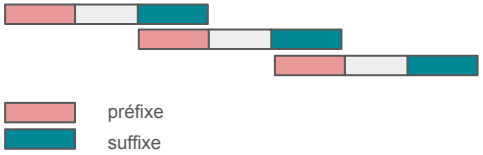
```python
def genome_path_string(kmers):
    """Reconstruct a string from its genome path.

    Input: list form : A sequence path of k-mers Pattern1, … ,
                       Patternn such that the last k - 1 symbols of Patterni
                       are equal to the first k-1 symbols of Patterni+1 for 1 ≤ i ≤ n-1.
    Output: A string Text of length k+n-1 such that the i-th k-mer in Text is equal to Patterni (for 1 ≤ i ≤ n).
    """
    # Start with the first k-mer
    genome_string = kmers[0]

    # Append the last character of each subsequent k-mer
    for kmer in kmers[1:]:
        genome_string += kmer[-1]

    return genome_string
```

```python
# Test
list_kmers = [
    "ACCGA",
    "CCGAA",
    "CGAAG",
    "GAAGC",
    "AAGCT",
]  # sequence path of k-mers need to be in order
result = genome_path_string(list_kmers)
pprint.pp(result)  # ACCGAAGCT
```

✓ 0.0s

Script 3

```python

def Prefix(Pattern):
    """Returns the prefix of a k-mer"""
    return Pattern[:-1]


def Suffix(Pattern):
    """Returns the suffix of a k-mer"""
    return Pattern[1:]

```



préfixe

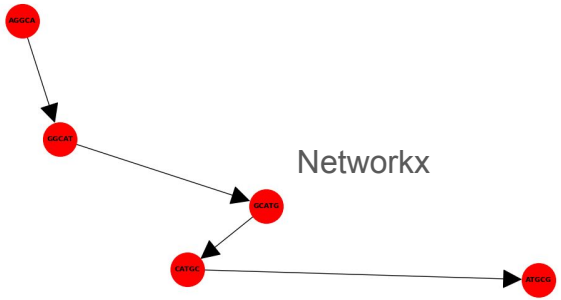
suffixe

```python
def Overlap(Patterns):
    """Construct the overlap graph of a collection of k-mers.

    :param Patterns: A collection Patterns of k-mers.
    :return: The overlap graph Overlap(Patterns), in the form of an adjacency list.
    """
    # Initialize an empty dictionary to store the adjacency list
    adj_list = {}

    # Iterate over each k-mer in Patterns
    for i, Pattern in enumerate(Patterns):
        # i is the index of the current k-mer in Patterns ; Pattern is the current k-mer
        # Iterate over each k-mer in Patterns
        for j, Pattern_prime in enumerate(Patterns):
            # j is the index of the current k-mer in Patterns ; Pattern_prime is the current k-mer
            # Skip the current k-mer
            if i == j:
                # if the index of the current k-mer is equal to the index of the current k-mer
                continue  # skip the current k-mer

            # Check if Suffix(Pattern) is equal to Prefix(Pattern_prime)
            if Suffix(Pattern) == Prefix(Pattern_prime):
                if Pattern not in adj_list:  # if Pattern is not in the adjacency list
                    adj_list[Pattern] = []  # add Pattern to the adjacency list
                adj_list[Pattern].append(Pattern_prime)
                # add Pattern_prime to the adjacency list of Pattern

    return adj_list  # return the adjacency list
```

```python
testseq = ["ATGCG", "GCATG", "CATGC", "AGGCA", "GGCAT"]
pprint.pp (Overlap (testseq))
```

✓ 0.0s

```
{'GCATG': ['CATGC'], 'CATGC': ['ATGCG'], 'AGGCA': ['GGCAT'], 'GGCAT': ['GCATG']}
```



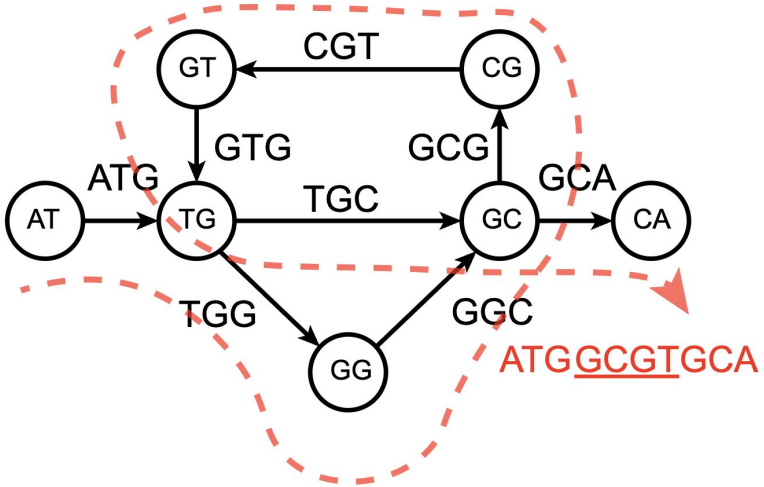
Networkx

Introduction

Modélisation du problème : intérêt des graphes

Graphe dirigé :

- Chaque k-mer définit une connexion dirigée
  - Préfixe → Suffixe
  - Exemple : pour "**CGT**" : Préfixe "CG" → Suffixe "GT"
- Nœuds = préfixes/suffixes de taille k-1
- Arêtes = k-mers complets
- Direction = ordre de succession dans la séquence



alexeidrummond.org

But :

- Parcourir chaque k-mer une fois
- → Recherche d'un chemin eulérien
- → Reconstruction de la séquence originale !

Script 5 *

```python
def CompositionGraph(Patterns):
    """
    Constructs the CompositionGraph of the collection of k-mers Patterns.
    :param Patterns: A collection of k-mers.
    :return: The composition graph CompositionGraph(Patterns),
    in the form of a tuple containing (Prefixes, Suffixes).
    """
    # Initialize an empty list to store the edges
    edges = []
    for kmer in Patterns:
        prefix = Prefix(kmer)
        suffix = Suffix(kmer)
        edges.append((prefix, suffix))
    return edges
```

```python
# Test the CompositionGraph function
listKmer = ["GAGG", "CAGG", "GGGG", "GGGA", "CAGG", "AGGG", "GGAG"]
pprint.pp(CompositionGraph(listKmer))
```

✓ 0.0s

```
[('GAG', 'AGG'),
 ('CAG', 'AGG'),
 ('GGG', 'GGG'),
 ('GGG', 'GGA'),
 ('CAG', 'AGG'),
 ('AGG', 'GGG'),
 ('GGA', 'GAG')]
```

Script 5 *

```
# Test the CompositionGraph function
listKmer = ["GAGG", "CAGG", "GGGG", "GGGA", "CAGG", "AGGG", "GGAG"]
pprint.pp(CompositionGraph(listKmer))
```

✓ 0.0s

```
[('GAG', 'AGG'),
 ('CAG', 'AGG'),
 ('GGG', 'GGG'),
 ('GGG', 'GGA'),
 ('CAG', 'AGG'),
 ('AGG', 'GGG'),
 ('GGA', 'GAG')]
```

```
def DeBruijn(Patterns):
    """
    Constructs the de Bruijn graph of a collection of k-mers Patterns.
    :param Patterns: A collection of k-mers.
    :return: The de Bruijn graph DeBruijn(Patterns), in the form of an adjacency list.
    """
    # Construct the CompositionGraph of Patterns
    edges = CompositionGraph(Patterns)
    # Initialize an empty dictionary to store the adjacency list
    adj_list = {}
    # Iterate over each edge in the CompositionGraph
    for edge in edges:
        # Unpack the edge into the prefix and suffix
        prefix, suffix = edge
        # Check if the prefix is already in the adjacency list
        if prefix in adj_list:
            # Append the suffix to the list of neighbors
            adj_list[prefix].append(suffix)
        else:
            # Initialize a new list with the suffix as the neighbor
            adj_list[prefix] = [suffix]
    # Sort the adjacency list by keys (prefixes) and values (suffixes) in alphabetical order
    sorted_adj_list = {k: sorted(v) for k, v in sorted(adj_list.items())}
    return sorted_adj_list
```

```
print("The De Bruijn graph of the given k-mers is:")
pprint.pp(DeBruijn(listKmer))
```

✓ 0.0s

```
The De Bruijn graph of the given k-mers is:
{'AGG': ['GGG'],
 'CAG': ['AGG', 'AGG'],
 'GAG': ['AGG'],
 'GGA': ['GAG'],
 'GGG': ['GGA', 'GGG']}
```

Cycle eulérien

```
# Eulerian Cycle Problem
"""
A cycle that traverses each edge of a graph exactly once is called an Eulerian cycle,
and we say that a graph containing such a cycle is Eulerian.
The following algorithm constructs an Eulerian cycle in an arbitrary directed graph.

    EULERIANCYCLE(Graph)
        form a cycle Cycle by randomly walking in Graph (don't visit the same edge twice!)
        while there are unexplored edges in Graph
            select a node newStart in Cycle with still unexplored edges
            form Cycle' by traversing Cycle (starting at newStart) and then randomly walking
            Cycle ← Cycle'
        return Cycle
"""

def EulerianCycle(Graph):
    """
    Finds an Eulerian cycle in a graph.
    param Graph: An Eulerian directed graph, in the form of an adjacency list.
    return: A list representing an Eulerian cycle in this graph.
    """

    # Initialize the cycle with any node from the graph. Here, we choose the first node in the keys of the adjacency list
    cycle = [list(Graph.keys())[0]]
    # Main loop that continues as long as there are edges in the graph to add to the cycle.
    while len(Graph) > 0:
        # When the current cycle is "closed" (the first and last nodes are the same),
        # check if this cycle contains a node with any unexplored edges.
        if cycle[0] == cycle[-1]:
            # As long as the first node of the cycle has no unexplored edges, rotate the cycle.
            # This means removing the first element of the cycle and appending it at the end.
            while cycle[0] not in Graph:
                cycle.pop(0)  # Remove the first node from the cycle.
                cycle.append(cycle[0])
                # Append this node at the end to "rotate" the cycle.
        # The last node of the current cycle becomes the new "starting point" or source to continue the traversal.
        source = cycle[-1]
        # Take one of the neighbors (or targets) of the current node in the graph (chosen randomly) and add it to the cycle.
        cycle.append(Graph[source].pop())
        # If the source node has no more neighbors (all its edges have been traversed), remove it from the graph. (clean up empty dict entries of graph)
        if len(Graph[source]) == 0:  # If the node has no more neighbors
            del Graph[source]  # delete the node from the graph
    return "->".join(cycle)  # Return the Eulerian cycle as a string

```

Algorithme général

Graphe = dictionnaire d'adjacence Python

Nœuds et arêtes représentant les k-mers

Consommation des arêtes durant le parcours

Construction du cycle par progression itérative depuis un noeud initial.

Gestion des sous-cycles

Cycle eulérien

Définition de la fonction et initialisation

```python
def EulerianCycle(Graph):
    """
    Finds an Eulerian cycle in a graph.
    Graph: An Eulerian directed graph, in the form of an adjacency list.
    Return: A list representing an Eulerian cycle in this graph.
    """



    # Data structure : dictionnaire d'adjacence
    Graph = {
    'AT': ['TG', 'TC'],
    'TG': ['GC', 'GT'],
    'TC': ['CG']
    }

    # Initialize the cycle with any node from the graph. Here, we choose the first node in the keys of the adjacency list
    cycle = [list(Graph.keys())[0]]
```

Structure de données = graphe représenté par une liste d'adjacence (dictionnaire Python tel que clés = nœuds (préfixes) et valeurs = liste des nœuds accessibles (suffixes))

Exemple: ici à l'initialisation, l'output de cycle = ['AT']

Cycle eulérien

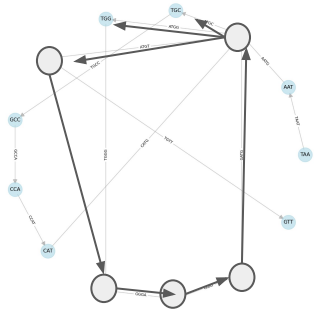
Boucle principale

```
    # Initialize the cycle with any node from the graph. Here, we choose the first node in the keys of the adjacency list
    cycle = [list(Graph.keys())[0]]

    # Main loop that continues as long as there are edges in the graph to add to the cycle.
    while len(Graph) > 0:

        # The last node of the current cycle becomes the new "starting point" or source to continue the traversal.
        source = cycle[-1]

        # Take one of the neighbors (or targets) of the current node in the graph (chosen randomly) and add it to the cycle.
        cycle.append(Graph[source].pop())

        # If the source node has no more neighbors (all its edges have been traversed), remove it from the graph. (clean up empty
        if len(Graph[source]) == 0:  # If the node has no more neighbors
            del Graph[source]  # delete the node from the graph
```

Par itération :

- Partir du dernier nœud visité
- Choisir une arête non visitée
- la supprimer simultanément du graphe

→ progression jusqu'à un graphe vide



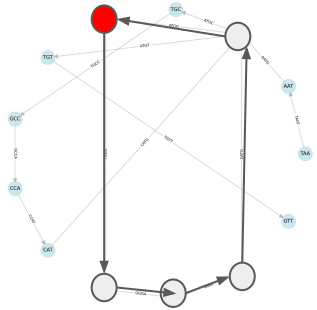
Cycle eulérien

Cycles fermés

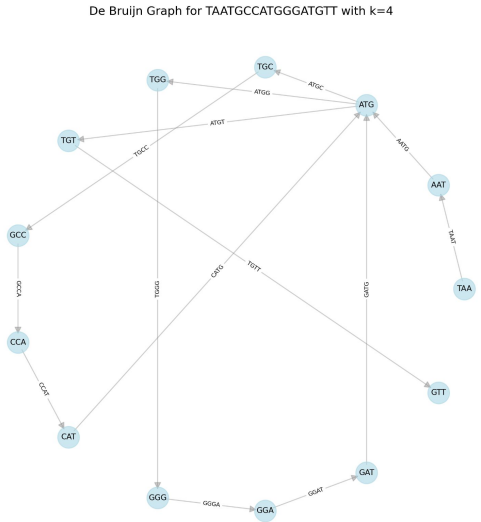
```
24          # When the current cycle is "closed" (the first and last nodes are equal),
25          # check if this cycle contains a node with any unexplored edges.
26          if cycle[0] == cycle[-1]:
27
28              # As long as the first node of the cycle has no unexplored edges, rotate the cycle.
29              # This means removing the first element of the cycle and appending it at the end.
30
31              while cycle[0] not in Graph:
32                  cycle.pop(0)  # Remove the first node from the cycle.
33                  cycle.append(cycle[0]) # Append this node at the end to "rotate" the cycle.
```

Graphe = { : ~~xy,~~
: yz,
: ab
}

cycle = [ ]



eulerien cycle

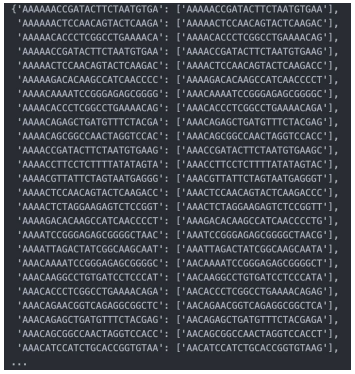


```python
# Fonction finale
def StringReconstruction(filename):   # filename is the path to the file containing the k-mers
    Patterns = read_kmers_from_file(filename)  # read the k-mers from the file
    dB = DeBruijn(Patterns)  # construct the de Bruijn graph # script 5
    path = EulerianPath(dB)  # find an Eulerian path in the de Bruijn graph #script 7
    Text = PathToGenome(path)  # reconstruct the string from the path #script 2
    return Text # sequence output
```

```
25
GTATAACGCGGTTACCGTAAAGACA
TAATTCAAATTAAAAAACTCCAACA
CAGGTAGAATCCAGCGGAGACCTTT
AGACAGTGAAACAGAACGGTCAGAG
GAAGGGACTATGCCCGCCTCCTGAA
CCAACTAGGTCCACCTTGCGCGGAA
CAGCTGGGTAATGGGTGCCATAAAT
TGGATACGAACCGATCAAAGATTCT
TCGGCGGGTGACACCAAGTACTAGC
CTCGCCGGCATATCAGACCTATCGT
TACGTGCGGCCGCGATAATTCAAAT
AAGATTCTGCTCACCCCATCTCGGG
AAAGACAGTGAAACAGAACGGTCAG
AGGTGTTCTTGCAATCAGTTTTTCT
CCATATGCCGCACGCGAAAACGTTT
TGAGTGCAAAATTAGACTATCGGCA
GGAATCAGGTGTTCTTGCAATCAGT
CAATATGCCATAAGGAATCCAACTG
GGCCAACTAGGTCCACCTTGCGCGG
TTCCTCTTTTATATAGTACCAGATT
AACAGTCAGGAGATGGGGTTAATCA
TTGCCAGCATTCGTGACCGAGCAGG
GCCGCAATGTAATTTCAGTTCCCGA
```

```
['GTATAACGCGGTTACCGTAAAGACA',
 'TAATTCAAATTAAAAAACTCCAACA',
 'CAGGTAGAATCCAGCGGAGACCTTT',
 'AGACAGTGAAACAGAACGGTCAGAG',
 'GAAGGGACTATGCCCGCCTCCTGAA',
 'CCAACTAGGTCCACCTTGCGCGGAA',
 'CAGCTGGGTAATGGGTGCCATAAAT',
 'TGGATACGAACCGATCAAAGATTCT',
 'TCGGCGGGTGACACCAAGTACTAGC',
 'CTCGCCGGCATATCAGACCTATCGT',
 'TACGTGCGGCCGCGATAATTCAAAT',
 'AAGATTCTGCTCACCCCATCTCGGG',
 'AAAGACAGTGAAACAGAACGGTCAG',
 'AGGTGTTCTTGCAATCAGTTTTTCT',
 'CCATATGCCGCACGCGAAAACGTTT',
 'TGAGTGCAAAATTAGACTATCGGCA',
 'GTTATTTATGCGCTTGTGGCTCCTA',
 'GGAATCAGGTGTTCTTGCAATCAGT',
 'CAATATGCCATAAGGAATCCAACTG',
 'GGCCAACTAGGTCCACCTTGCGCGG',
 'TACAGACATCGCATTTCATGAGAGG',
 'CCTCCTGAACTATGTGGCAATGCCA',
 'TTCCTCTTTTATATAGTACCAGATT',
 'AACAGTCAGGAGATGGGGTTAATCA',
 'TTGCCAGCATTCGTGACCGAGCAGG',
 ...
```

```
[('GTATAACGCGGTTACCGTAAAGAC', 'TATAACGCGGTTACCGTAAAGACA'),
 ('TAATTCAAATTAAAAAACTCCAAC', 'AATTCAAATTAAAAAACTCCAACA'),
 ('CAGGTAGAATCCAGCGGAGACCTT', 'AGGTAGAATCCAGCGGAGACCTTT'),
 ('AGACAGTGAAACAGAACGGTCAGA', 'GACAGTGAAACAGAACGGTCAGAG'),
 ('GAAGGGACTATGCCCGCCTCCTGA', 'AAGGGACTATGCCCGCCTCCTGAA'),
 ('CCAACTAGGTCCACCTTGCGCGGA', 'CAACTAGGTCCACCTTGCGCGGAA'),
 ('CAGCTGGGTAATGGGTGCCATAAA', 'AGCTGGGTAATGGGTGCCATAAAT'),
 ('TGGATACGAACCGATCAAAGATTC', 'GGATACGAACCGATCAAAGATTCT'),
 ('TCGGCGGGTGACACCAAGTACTAG', 'CGGCGGGTGACACCAAGTACTAGC'),
 ('CTCGCCGGCATATCAGACCTATCG', 'TCGCCGGCATATCAGACCTATCGT'),
 ('TACGTGCGGCCGCGATAATTCAAA', 'ACGTGCGGCCGCGATAATTCAAAT'),
 ('AAGATTCTGCTCACCCCATCTCGG', 'AGATTCTGCTCACCCCATCTCGGG'),
 ('AAAGACAGTGAAACAGAACGGTCA', 'AAGACAGTGAAACAGAACGGTCAG'),
 ('AGGTGTTCTTGCAATCAGTTTTTC', 'GGTGTTCTTGCAATCAGTTTTTCT'),
 ('CCATATGCCGCACGCGAAAACGTT', 'CATATGCCGCACGCGAAAACGTTT'),
 ('TGAGTGCAAAATTAGACTATCGGC', 'GAGTGCAAAATTAGACTATCGGCA'),
 ('GTTATTTATGCGCTTGTGGCTCCT', 'TTATTTATGCGCTTGTGGCTCCTA'),
 ('GGAATCAGGTGTTCTTGCAATCAG', 'GAATCAGGTGTTCTTGCAATCAGT'),
 ('CAATATGCCATAAGGAATCCAACT', 'AATATGCCATAAGGAATCCAACTG'),
 ('GGCCAACTAGGTCCACCTTGCGCG', 'GCCAACTAGGTCCACCTTGCGCGG'),
 ('TACAGACATCGCATTTCATGAGAG', 'ACAGACATCGCATTTCATGAGAGG'),
 ('CCTCCTGAACTATGTGGCAATGCC', 'CTCCTGAACTATGTGGCAATGCCA'),
 ('TTCCTCTTTTATATAGTACCAGAT', 'TCCTCTTTTATATAGTACCAGATT'),
 ('AACAGTCAGGAGATGGGGTTAATC', 'ACAGTCAGGAGATGGGGTTAATCA'),
 ('TTGCCAGCATTCGTGACCGAGCAG', 'TGCCAGCATTCGTGACCGAGCAGG'),
 ...
```



```
{'AAAAAACCGATACTTCTAATGTGA': ['AAAAACCGATACTTCTAATGTGAA'],
 'AAAAAACTCCAACAGTACTCAAGA': ['AAAAACTCCAACAGTACTCAAGAC'],
 'AAAAACACCCTCGGCCTGAAAACA': ['AAAACACCCTCGGCCTGAAAACAG'],
 'AAAAACCGATACTTCTAATGTGAA': ['AAAACCGATACTTCTAATGTGAAG'],
 'AAAAACTCCAACAGTACTCAAGAC': ['AAAACTCCAACAGTACTCAAGACC'],
 'AAAAAGACACAAGCCATCAACCCC': ['AAAAGACACAAGCCATCAACCCCT'],
 'AAAACAAAATCCGGGAGAGCGGGG': ['AAACAAAATCCGGGAGAGCGGGGC'],
 'AAAACACCCTCGGCCTGAAAACAG': ['AAACACCCTCGGCCTGAAAACAGA'],
 'AAAACAGAGCTGATGTTTCTACGA': ['AAACAGAGCTGATGTTTCTACGAG'],
 'AAAACAGCGGCCAACTAGGTCCAC': ['AAACAGCGGCCAACTAGGTCCACC'],
 'AAAACCGATACTTCTAATGTGAAG': ['AAACCGATACTTCTAATGTGAAGC'],
 'AAAACCTTCCTCTTTTATATAGTA': ['AAACCTTCCTCTTTTATATAGTAC'],
 'AAAACGTTATTCTAGTAATGAGGG': ['AAACGTTATTCTAGTAATGAGGGT'],
 'AAAACTCCAACAGTACTCAAGACC': ['AAACTCCAACAGTACTCAAGACCC'],
 'AAAACTCTAGGAAGAGTCTCCGGT': ['AAACTCTAGGAAGAGTCTCCGGTT'],
 'AAAAGACACAAGCCATCAACCCCT': ['AAAGACACAAGCCATCAACCCCTG'],
 'AAAATCCGGGAGAGCGGGGCTAAC': ['AAATCCGGGAGAGCGGGGCTAACG'],
 'AAAATTAGACTATCGGCAAGCAAT': ['AAATTAGACTATCGGCAAGCAATA'],
 'AAACAAAATCCGGGAGAGCGGGGC': ['AACAAAATCCGGGAGAGCGGGGCT'],
 'AAACAAGGCCTGTGATCCTCCCAT': ['AACAAGGCCTGTGATCCTCCCATA'],
 'AAACACCCTCGGCCTGAAAACAGA': ['AACACCCTCGGCCTGAAAACAGAG'],
 'AAACAGAACGGTCAGAGGCGGCTC': ['AACAGAACGGTCAGAGGCGGCTCA'],
 'AAACAGAGCTGATGTTTCTACGAG': ['AACAGAGCTGATGTTTCTACGAGA'],
 'AAACAGCGGCCAACTAGGTCCACC': ['AACAGCGGCCAACTAGGTCCACCT'],
 'AAACATCCATCTGCACCGGTGTAA': ['AACATCCATCTGCACCGGTGTAAG'],
 ...
```

```
['GGCATAGGCTCGTCCTCTCATCAC',
 'GCATAGGCTCGTCCTCTCATCACT',
 'CATAGGCTCGTCCTCTCATCACTG',
 'ATAGGCTCGTCCTCTCATCACTGG',
 'TAGGCTCGTCCTCTCATCACTGGT',
 'AGGCTCGTCCTCTCATCACTGGTT',
 'GGCTCGTCCTCTCATCACTGGTTT',
 'GCTCGTCCTCTCATCACTGGTTTC',
 'CTCGTCCTCTCATCACTGGTTTCT',
 'TCGTCCTCTCATCACTGGTTTCTG',
 'CGTCCTCTCATCACTGGTTTCTGG',
 'GTCCTCTCATCACTGGTTTCTGGA',
 'TCCTCTCATCACTGGTTTCTGGAT',
 'CCTCTCATCACTGGTTTCTGGATG',
 'CTCTCATCACTGGTTTCTGGATGT',
 'TCTCATCACTGGTTTCTGGATGTT',
 'CTCATCACTGGTTTCTGGATGTTG',
 'TCATCACTGGTTTCTGGATGTTGC',
 'CATCACTGGTTTCTGGATGTTGCT',
 'ATCACTGGTTTCTGGATGTTGCTA',
 'TCACTGGTTTCTGGATGTTGCTAT',
 'CACTGGTTTCTGGATGTTGCTATC',
 'ACTGGTTTCTGGATGTTGCTATCT',
 'CTGGTTTCTGGATGTTGCTATCTA',
 'TGGTTTCTGGATGTTGCTATCTAC',
 ...
```

'GGCATAGGCTCGTCCTCTCATCACTGGTTTCTGGATGTTGCTATCTACTCATTTTGTTGTGGCTCAGCCTACTTTGCCGCAATGTAATTTCAGTTCCCGATCCAGCTGGGTAATGGGTGCCATAAATTTCCAACCAACACTTGGGCACGGCGGCAG
TCTCGGAGAAGCCTACGAAACATCCATCTGCACCGGTGTAAGGGCAAGCCTGAGTGCAAAATTAGACTATCGGCAAGCAATAATTAAGACTTTCGCAAGGCATGAGGGAGTCTATTGGATTTCCCTGTATCGTATTCTAGGACGATTTACGGATTC
ATACGACATTTCCACTGACTGAAGCTCAGTACAGGTCGCGTCCTACAATCTAACCAATGGTAGGGCATATCTAAATTTGCTGCATAAAACCTTCCTCTTTTATATAGTACCAGATTATGATTAAGGCTCCGCTACCCGCCTCTGTTATTTATGCGCTT
GTGGCTCCTAGGAACAGTCAGGAGATGGGGTTAATCACGTGTGACTCGCCGAGGTGAAATAGCGCACCTATTTCATGTCTGCTGACTCTGGTGTGGTACTCAATACGTTATGCCAATTGCATCCTCCGACGACGTTAGTAGAATAACTCAATTTGT
GAACAGTAATCAGGTCTAAACAAGGCCTGTGATCCTCCCATAAGTGGTCCGGGAAGTGCTCTGTCCGACCTTGCTCAATATCACCACGCAATATGCCATAAGGAATCCAACTGGGACCTGGCACGCCTCGCCGGCATATCAGACCTATCGTCGCG
TTCCTATAAAACAAAATCCGGGAGAGCGGGGCTAACGCGCACTCAAAAAGACACAAGCCATCAACCCCTGATTTACGAGCGCGATGCTCCCTCGCCCATCCAAGCGCACCAGAGGATACGGGACTCCATCTTAAAGCCAGAAGGGCGACTGCAC
CGGCCCCAGTACGGTGAAAACAGCGGCCAACTAGGTCCACCTTGCGCGGAATCCGGCGGACAGGTGAATACGTGCGGCCGCGATAATTCAAATTAAAAAACTCCAACAGTACTCAAGACCCGTCATCTTTAAGAAACGAAGCCGACCTTTTGCCT
AGTCGCTCAGTGAGGACTAGGGTTATTGTCTTATGGTTCACCAAGCCTTGCAATGTATCCTCTCCGTTGCGCTGTGGTCAGATGACGAAGAACAGGTAGAATCCAGCGGAGACCTTTACACGTAGAAGCAAAAAACCGATACTTCTAATGTGAAGC
TACACAAGACGGATAGTTTACGGATAAGTACCTATCGTTATTCATCATCTACGCAACCCACGCTGCGCCGCTGACACTGAGTTACTTATGACGTCTTCGGGGTGAAAACTCTAGGAAGAGTCTCCGGTTAGCCATCTGCCCCATCAGGTTAATGTG
AAGACTTGGCGCTCGGAATCAGGTGTTCTTGCAATCAGTTTTTCTCGGGTGCGATTAAAGTCCAGAGGTCTTTGGGAATGTGACCAAAGATTGCGTTACTGCTATTTCGCGTTGTCCAATGGTAGATACCTTGCCAGCATTCGTGACCGAGCAGGC
CTGTTGACGAGGCTGGTTTACGAGGATCAATCGATACCATATGCCCGCACGCGAAAACGTTATTCTAGTAATGAGGGTTAGAGACCGGAAGGGACTATGCCCGCCTCCTGAACTATGTGGCAATGCCATCAAATGCTACTGCCAGTCCCGATCTT
TGGAAGTGGGTCCACACCTACAGACATCGCATTTCATGAGAGGGCTGGGGCAGGGTTGGATACGAACCGATCAAAGATTCTGCTCACCCCATCTCGGGGCTACGGTGTATGAACATGTGTCGTAAGCTCAGGCGTTGCTCATTTTGTCCTTTAAG
GTTACTGGAGGACTTTCGCGTATAACGCGGTTACCGTAAAGACAGTGAAACAGAACGGTCAGAGGCGGCTCAACGAGTACCTTGGGCTCGCGGGCACTATCCCCTGATTATTCTCATGATGTCATCTTCTGTTCGCTATAACTGTCGGAGTATCAT
GTCAACGCAGTAATCAGCTCTAAAAACACCCTCGGCCTGAAAACAGAGCTGATGTTTCTACGAGAGATGGTACATCCTAGAGGGCCCGGACAACTACATCTCACGGTAGTATGTGCTTCGGCGGGTGACACCAAGTACTAGCCAAATGTTCGAAC
GTGCGACAGGCTCTGTCTAGGCACCTCGCGTGCTATAGCTAGTCAAGGCAAGCCTCGAGTG'

https://alexeidrummond.org/bayesian_phylo_lectures/lectureGenomeAssembly/?print-pdf#/

git@github.com:hakimGMO/Team-PLS-Genome-Assembly.git

https://www.bioinformaticsalgorithms.org/bioinformatics-chapter-3

Script 4

```python
def PathGraphk(Text, k):
    """
    Generate the nodes and edges of the path graph from a genome Text.

    PathGraphk(Text) is the path consisting of |Text| - k + 1 edges,
    where the i-th edge of this path is labeled by the i-th k-mer in Text and the i-th node of
    the path is labeled by the i-th (k - 1)-mer in Text.

    :param Text: A string representing the genome.
    :param k: An integer representing the length of k-mers.

    :return: A tuple containing two lists:
        - nodes: A list of (k-1)-mers representing the nodes of the path graph.
        - edges: A list of k-mers representing the edges of the path graph.
    """
    # Initialize an empty list to store the nodes
    nodes = []
    # Initialize an empty list to store the edges
    edges = []

    # Iterate over the range from 0 to the length of Text - k+1
    for i in range(len(Text) - k + 1):
        # Append the i-th (k - 1)-mer in Text to the nodes list
        nodes.append(Text[i : i + k - 1])
        # Append the i-th k-mer in Text to the edges list
        edges.append(Text[i : i + k])

    return nodes, edges
```

```python
# Test the PathGraphk function
k1 = 4  # length of k-mers
text1 = "AAGATTCTCTAC"  # genome Text

pprint.pp(PathGraphk(text1, k1)) #
```

✓ 0.0s

```
(['AAG', 'AGA', 'GAT', 'ATT', 'TTC', 'TCT', 'CTC', 'TCT', 'CTA'],
 ['AAGA', 'AGAT', 'GATT', 'ATTC', 'TTCT', 'TCTC', 'CTCT', 'TCTA', 'CTAC'])
```

Script 4

```python
def DeBruijnk(Text, k):
    """
    Generate the adjacency list of the de Bruijn graph from a genome Text.

    The de Bruijn graph DeBruijnk(Text) is formed by gluing identically labeled nodes in PathGraphk(Text).

    :param Text: A string representing the genome.
    :param k: An integer representing the length of k-mers.

    :return: A dictionary representing the adjacency list of the de Bruijn graph.
    """
    # Generate the nodes and edges of the path graph nodes,
    edges = PathGraphk(Text, k)

    # Initialize an empty dictionary to store the adjacency list
    adj_list = {}

    # Iterate over each edge in the edges list
    for i in range(len(edges[1])):  # edges[1] is the list of k-mers
        edge = edges[1][i]  # get the i-th k-mer
        # Get the prefix of the edge
        prefix = edge[: k - 1]
        # Get the suffix of the edge
        suffix = edge[1:]

        # If the prefix is not in the adjacency list, add it
        if prefix not in adj_list:
            adj_list[prefix] = []

        # Add the suffix to the adjacency list for the prefix
        adj_list[prefix].append(suffix)

    return adj_list
```

```python
# Test the DeBruijnk function
result = DeBruijnk(text1, k1)
pprint.pp(result)
```

✓ 0.0s

```
{'AAG': ['AGA'],
 'AGA': ['GAT'],
 'GAT': ['ATT'],
 'ATT': ['TTC'],
 'TTC': ['TCT'],
 'TCT': ['CTC', 'CTA'],
 'CTC': ['TCT'],
 'CTA': ['TAC']}
```