**Explanation of Functions**

Please refer to this page if you need clarification about functions. It provides steps and examples of intended code. Contains version 1 and at the end, version 2 with changes using deque function.

**VERSION 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

A. **debrujin\_graph\_from\_kmers(patterns) function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

This code defines a function called debrujin\_graph\_from\_kmers which takes a list of DNA sequences represented as strings called patterns as input.

The goal of the function is to generate a de Bruijn graph from the given list of DNA sequences.

The function starts by creating a list called kmers that contains all the suffixes of each input sequence by calling a helper function called suffix\_composition.

The suffix\_composition function generates all the suffixes of a given length k from a given pattern string, and the uniq parameter ensures that duplicates are removed from the resulting list of suffixes. The kmers list is then converted to a set to remove duplicates.

Next, the function creates an empty dictionary called dict that will be used to represent the de Bruijn graph. The keys of the dictionary are the kmers generated from the input sequences, and their values are initially set to empty lists.

The function then iterates over each input sequence kmer and adds an entry to the dict dictionary. The key of the entry is the prefix of the kmer, and the value is the suffix of the kmer. This effectively builds the edges of the de Bruijn graph.

Finally, the function returns the dict dictionary, which represents the de Bruijn graph constructed from the input sequences.

Note that the prefix and suffix functions used to extract the prefix and suffix mentioned in the code.

Use duque function.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Example DNA sequences

patterns = ['ATCGCGATCG', 'CGATCGCGAT', 'GATCGCGATC']

# Construct de Bruijn graph

graph = debrujin\_graph\_from\_kmers(patterns)

# Print de Bruijn graph

for key, value in graph.items():

print(key + ' -> ' + ‘,’.join(value))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this example, the list patterns contains three DNA sequences: 'ATCGCGATCG', 'CGATCGCGAT', and 'GATCGCGATC'.

We call the debrujin\_graph\_from\_kmers function with patterns as the argument, and it returns a dictionary that represents the de Bruijn graph of the input sequences.

To visualize the graph, we iterate over each key-value pair in the dictionary and print them out in the format key -> value1,value2,…,valueN.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The output of the code should be:

AT -> TCG

CG -> GAT

GA -> ATC

TC -> CGA

CG -> GCG

GC -> CGA

AT -> TCG

CG -> GAT

CG -> GCG

GA -> ATC

AT -> TCG

CG -> GCG

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This output shows that the de Bruijn graph has six nodes (AT, CG, GA, TC, CG, and GC) and twelve edges. Each key in the dictionary corresponds to a node in the graph, and its values correspond to the outgoing edges from that node. For example, the key 'AT' has a single outgoing edge to the node 'TCG'.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**B. genomePath(kmers, apppend\_last=True) function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

This code defines a function named "genomePath" that takes two arguments: "kmers", which is a list of strings representing DNA k-mers, and "append\_last", which is a boolean value with a default value of True.

The function iterates through each k-mer in the list "kmers", extracts the first character of each k-mer, and concatenates these characters to form a new string “genome".

If the "append\_last" argument is True, the function also adds the remaining characters of the last k-mer in the list "kmers" to the end of the "genome" string.

This is done using the string slicing notation [1:], which extracts all characters in the k-mer except for the first character.

Finally, the function returns the "genome" string. Therefore, this function takes a list of DNA k-mers and returns a single string representing the reconstructed DNA sequence that consists of overlapping k-mers.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Example DNA k-mers

kmers = ['ACGT', 'CGTT', 'GTTA', 'TTAC']

# Reconstruct DNA sequence

dna\_seq = genomePath(kmers)

# Print reconstructed DNA sequence

print(dna\_seq)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this example, the list "kmers" contains four DNA k-mers: 'ACGT', 'CGTT', 'GTTA', and 'TTAC'.

These k-mers overlap by k-1 nucleotides, meaning that the last nucleotide of each k-mer matches the first nucleotide of the next k-mer.

When we call the "genomePath" function with the "kmers" list as the argument, it returns a single string representing the reconstructed DNA sequence, which is ‘ACGTTTAC'.

Note that by default, the "append\_last" argument is True, so the last k-mer 'TTAC' is included in the reconstructed sequence. If we set "append\_last" to False, the function would not include the last k-mer in the sequence, and the reconstructed sequence would be 'ACGTT'.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**C.** eulPath(dict)**function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

The function eulPath(dict) takes a dictionary representation of the

De Bruijn graph as input and returns the Eulerian path through the graph as a

list of nodes.

This is done by iteratively traversing the graph and adding nodes

to a stack until a dead end is reached, at which point the last node is added

to the path and removed from the stack.

Here is what the code does:

1. It initializes an empty stack and a dictionary called balanced\_count using the balanceCount function.
2. It adds a node to the stack, which is the node with an imbalance of -1 (i.e., the node that has one more outgoing edge than incoming edges).
3. It initializes an empty list called path.
   1. It enters a while loop that continues until the stack is empty. Within the loop, it performs the following steps:
   2. It retrieves the top of the stack u\_v.
   3. It attempts to retrieve the first adjacent node w of u\_v from the dict dictionary.
   4. If w is found, it is added to the stack and removed from dict[u\_v]. If w is not found, the node u\_v has no remaining unexplored edges, so u\_v is added to the path list and removed from the stack.
4. The path list is reversed using the slice notation [::-1] and returned.

Therefore, the eulPath function takes a dictionary dict representing a graph, and returns a list of nodes representing an Eulerian path (a path that visits every edge exactly once) in the graph. The function uses a stack to keep track of the current path and performs a depth-first search to explore all edges in the graph. The algorithm starts at a node with an imbalance of -1 (if one exists), which guarantees the existence of an Eulerian path.

Use duque function.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Define an example graph represented by a dictionary

graph = {'ACT': ['CTG'], 'CTG': ['TGA'], 'TGA': ['GAC'], 'GAC': ['ACT']}

# Find an Eulerian path in the graph using the eulPath function

path = eulPath(graph)

# Concatenate the nodes in the path to get the final DNA sequence

sequence = ''.join(path)

# Print the resulting sequence

print(sequence)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this example, we define a directed graph of DNA sequences, where each node represents a k-mer and each edge represents a suffix-prefix overlap between two k-mers.

We assume that the graph has an Eulerian path, and use the eulPath function to find a valid path through the graph. We then concatenate the nodes in the resulting path to obtain the final DNA sequence, which we store in the variable sequence. Finally, we print the resulting sequence.

Note that the eulPath function assumes that the input graph has an Eulerian path (i.e., it has exactly two nodes with an imbalance of +/- 1) and does not perform any error checking to ensure this. If the graph does not have an Eulerian path, the function may enter an infinite loop or produce unexpected results.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**D. suffix\_composition(k, text, uniq=False)function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

suffix\_composition(k, text, uniq=False), generates all k-length suffixes

of a given string

The Python code defines a function called suffix\_composition that takes three arguments: k, text, and uniq. Here is what the code does:

1. It initializes an empty list called kmers.
2. It iterates through the characters in text using a for loop. Specifically, it loops through range(len(text)+1-k). This is equivalent to looping through all possible starting positions of a substring of length k in the text.
3. For each iteration of the loop, the code extracts a substring of length k-1 starting from the current position in text and appends it to the kmers list.
4. If uniq is True, the function converts the kmers list to a set to remove duplicates, then back to a sorted list before returning it.
5. If uniq is False, the function sorts the kmers list and returns it.

Therefore, the suffix\_composition function takes a string text and an integer k and returns a list of all substrings of text of length k-1 (i.e., its suffixes) sorted in lexicographic order. If uniq is True, the list only contains unique elements.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

text = "ACGTTAGC"

k = 3

suffixes = suffix\_composition(k, text)

print(suffixes)

['ACG', 'AGC', 'CGT', 'GTA', 'GTT', 'TAG', ‘TTA’]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this example, we call the suffix\_composition function with the DNA sequence "ACGTTAGC" and a value of 3 for k.

The function returns a list of all substrings of length k-1 of the sequence "ACGTTAGC". The returned list is sorted in lexicographic order and stored in the variable suffixes. Finally, we print the contents of suffixes to the console, which shows the list of substrings ['ACG', 'AGC', 'CGT', 'GTA', 'GTT', 'TAG', ‘TTA'].

**E. balanceCount(adjacentList) function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

balanceCount(adjacentList), computes the in-degree and out-degree of

each node in the graph.

The balanceCount(adjacentList) function in Python takes a dictionary adjacentList as input, where the keys of the dictionary represent nodes in a graph, and the values of the dictionary are lists of nodes adjacent to the corresponding key node.

This function computes the balance count for each node in the graph, which is defined as the difference between the number of outgoing edges from a node and the number of incoming edges to that node.

The function returns a dictionary balanced\_count with the same keys as the input adjacentList dictionary, where each key is associated with its corresponding balance count value.

The function first initializes a new dictionary balanced\_count with the same keys as adjacentList, and sets the value for each key to 0. It then iterates through each node in adjacentList, and for each outgoing edge from that node, it decreases the balance count for the source node by 1 and increases the balance count for the target node by 1.

If the target node is not already in the balanced\_count dictionary, it adds it and sets its value to 1.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For example:

The balanced\_count dictionary has the same keys as adjacentList, and the values represent the balance count for each DNA sequence. The balance count is computed in the same way as before, by counting the difference between the number of outgoing edges and incoming edges for each node in the graph.

# Input adjacentList

adjacentList = {'ATG': ['TGC', 'GCA', 'CAT'],

'TGC': ['GCC', 'GCT'],

'GCC': ['CCA'],

'GCT': ['CTT'],

'CAT': ['ATG']}

# Output balanced\_count

{'ATG': 0, 'TGC': 0, 'GCC': 0, 'GCT': 0, 'CAT': 0, 'TGC': 1, 'GCA': -1, 'CAT': -1, 'GCC': 1, 'GCT': -1, 'CCA': 1}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**F.suffix(string)) function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

The **suffix(string) function** in Python takes a string as an argument and returns a new string consisting of all the characters in the input string except the first one.

This function achieves this by using slicing notation [1:], which specifies a slice of the input string that starts at the second character (index 1) and includes all the remaining characters to the end of the string.

For example, if you call the function with the argument "ACG", it will return "CG".

**G prefix(string): function**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

The **prefix(string) function** in Python takes a string as an argument and returns a new string consisting of all the characters in the input string except the last one.

This function achieves this by using slicing notation [0:-1], which specifies a slice of the input string that starts at the first character (index 0) and ends at the second-to-last character (index -1).

For example, if you call the function with the argument "ACG", it will return "AC".

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**VERSION 2**

Get to know deque:

“Appending and popping items from the right end of a Python list is usually

efficient (O(1)), but can become slower (O(n)) when the list needs to

reallocate memory. Appending and popping items from the left end of a list is

generally inefficient (O(n)). While Python lists can be used as stacks and

queues, their performance issues can impact overall application performance.

To address these issues, Python's deque data type was introduced, which provides

memory-efficient and fast append and pop operations on both ends of the data

structure.”

Updates:

## To optimize the **debrujin\_graph\_from\_kmers(patterns)** function, we can:

Use a defaultdict object from the collections module to replace the dict object.

This will automatically initialize keys with an empty deque object, enabling us

to use a deque to store the list of outgoing edges for each node in the graph.

Used the append() method to add each outgoing edge to the deque instead of

dict[prefix(kmer)].append(suffix(kmer)). This will append a new item to a deque

object instead of a Python list object, which can improve performance.

Changed the loop to add all suffixes to the set of kmers using

kmers.add(suffix(kmer)) instead of creating a new list kmers and concatenating

it with the suffix composition of each pattern. This approach will avoid

creating a list of unnecessary strings.

## To optimize the **eulPath(dict) function**, we can take the following steps:

This code is implementing the Eulerian Path algorithm for a directed graph. It utilizes the deque data structure from the collections module to optimize the algorithm's efficiency.

In the first line of the code, a deque object is initialized and named dictionary.

Then, a node with an indegree that is one more than its outdegree is added to the dictionary object using dictionary.appendleft(). This node will be used as the starting point of the Eulerian Path.

In the while loop, the algorithm repeatedly takes the leftmost node from the dictionary deque, gets its adjacent node from the graph, and appends it to the left side of the dictionary deque. This operation is done using popleft() and appendleft() methods, respectively.

If a node has no more adjacent nodes, it is removed from the dictionary deque and added to the path deque using path.appendleft().

Finally, the path deque is returned, which contains the nodes in the order they should be visited to complete the Eulerian Path.

Using a deque as the data structure for both dictionary and path allows the algorithm to efficiently remove and append elements from both ends of the deque. This optimization results in improved performance compared to using a standard list.