CSE 5370: BIO-INFORMATICS

HOMEWORK – 2

1) Shortest Common Superstring

1.1) Coding Explaination

The code defines a function overlap() that computes the overlap length between two input strings read_a and read_b. The function shortestCommonSuperstring() takes a list of input strings string_set and tries all permutations of the input strings to find the shortest common superstring that contains all input strings.

The function first generates an adjacency matrix adj that stores the overlap lengths between every pair of input strings using the overlap() function. It then generates all possible permutations of the indices of the input strings using itertools.permutations(). For each permutation, the function constructs a superstring by appending the non-overlapping portion of each consecutive pair of strings in the permutation to the previous string. The length of the resulting superstring is compared to the length of the shortest superstring found so far, and if the new superstring is shorter, it is stored as the new shortest superstring.

1.2) Reasoning About Time Complexity

The first code uses brute force approach to solve the problem, where all permutations of the input strings are generated and the shortest superstring is found by trying all the permutations. This involves calculating the overlap of every pair of strings in the list, which takes O(n^2 * m) time, where n is the number of strings and m is the maximum length of the strings. Then, it generates all permutations of the strings, which takes O(n!) time. Finally, for each permutation, it calculates the superstring by concatenating the strings in the order specified by the permutation, which takes O(n * m) time. Therefore, the total time complexity of the first code is O(n! * n^2 * m).

The second code uses a greedy approach to solve the problem, where it calculates the overlap of every pair of strings in the list and constructs an overlap graph. Then, it finds the shortest superstring by using a Hamiltonian path algorithm on the overlap graph. The time complexity of constructing the overlap graph is O(n^2 * m) and the time complexity of finding the Hamiltonian path is O(n^2). Therefore, the total time complexity of the second code is O(n^2 * m + n^2).

Comparing the two, the second code is more efficient, with a lower time complexity of O(n^2 * m + n^2). This is because it uses a greedy approach, where it makes locally optimal choices at each step, instead of trying all permutations. The

greedy approach is more efficient because it reduces the search space and avoids the need to generate and test all permutations.

I've tested the code by passing sample strings of my genome sequence and calculated it's time complexity and the following output is printed below comparing both the codes scsfast.py represents code2 and scsfo.py represents code 1 given in the question.

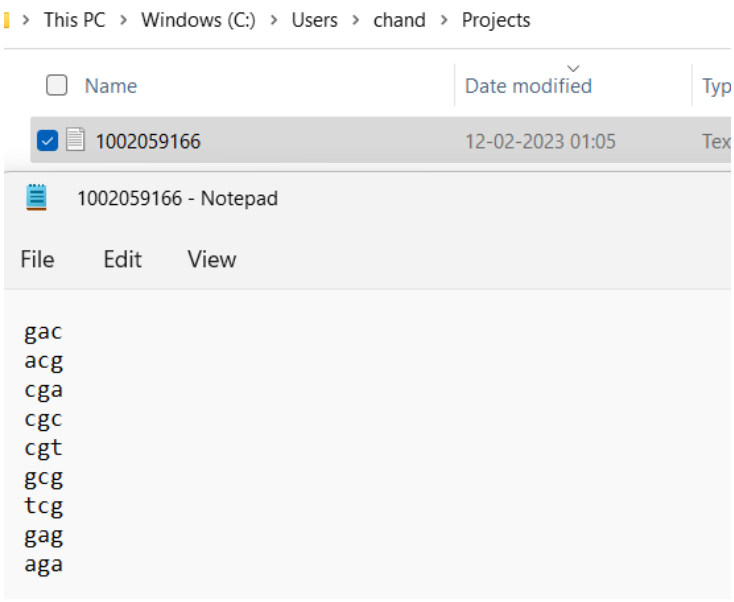
```
= RESTART: C:\Users\chand\Desktop\UTA-SEM\MS-SEM2\BIO-INFORMATICS\HW2-PROJECT\scsfo.py
TCGAGACGCGT
Elapsed time:  1.5689833164215088 seconds

= RESTART: C:\Users\chand\Desktop\UTA-SEM\MS-SEM2\BIO-INFORMATICS\HW2-PROJECT\scsfast.py
TCGAGACGCGT
Elapsed time:  0.6480646133422852 seconds
```

2) De Bruijn Graphs

Generating Your Own Unique Data

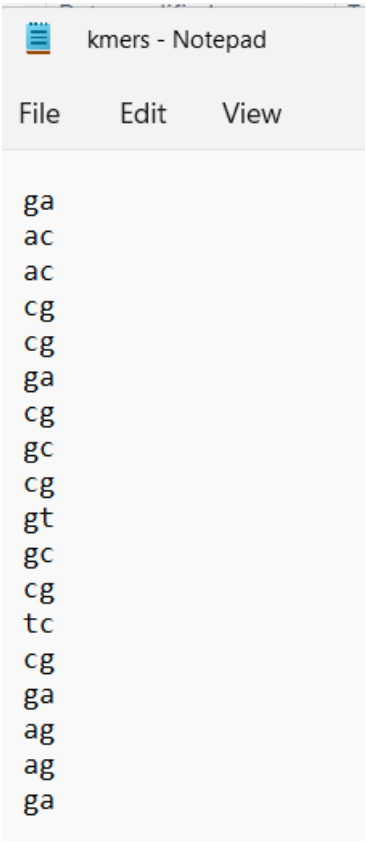
Ran the "datasetGenerator_hw2.py" script with my UTA ID and generated the 1002059166.txt file as shown below.

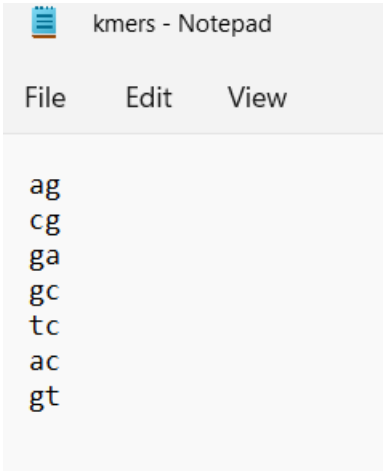
```
C:\Users\chand\Projects>python3 datasetGenerator_hw2.py --ID 1002059166

C:\Users\chand\Projects>
```



Generating K-mers

Generated the data for k=2 and stored in kmers.txt file as shown below. As per code I've generated only unique sets. But for understanding I've also included the screenshots of all possible kmers.

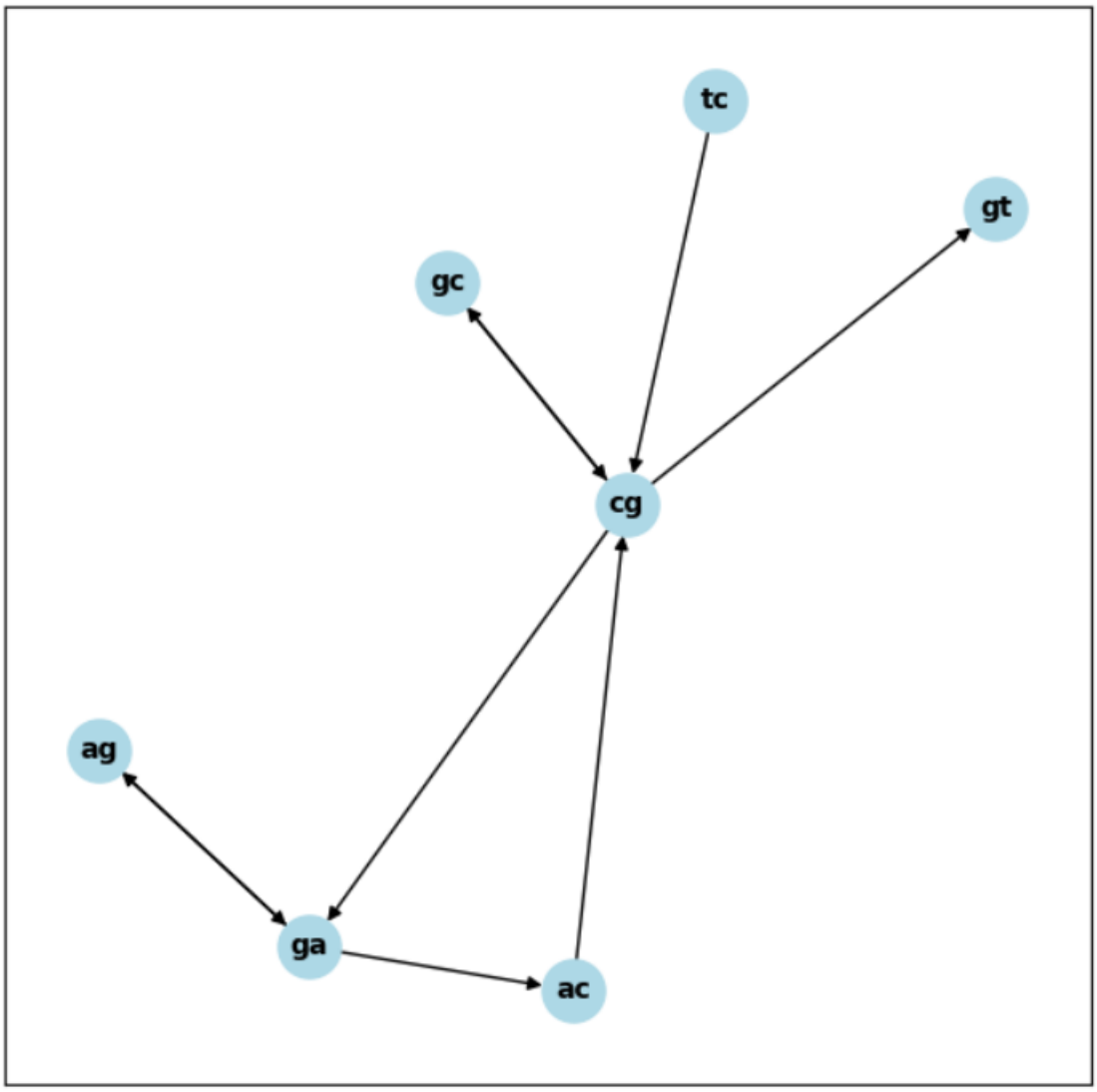




2.3) Generating a De Bruijn Graph

The below picture depicts the nodes and edges, through which graph is connected.

```
ga -> ac,ag
ac -> cg
cg -> ga,gc,gt
gc -> cg
tc -> cg
ag -> ga
```



2.4) Eulerian Cycles

If the De Bruijn graph does not have an Eulerian cycle, it is possible to find the minimum set of reads that could add a Eulerian cycle to the graph. To do this, we can look for nodes with an odd degree, which represent edges that need to be added to create an Eulerian cycle. The minimum set of reads would be the ones that connect these nodes with each other. Once these reads are added, we can then find an Eulerian cycle in the graph and obtain the genome sequence as described above.

In this event the De Bruijn graph has an Eulerian cycle, it is possible to develop genome sequence.

```
In-degree: {'ga': 2, 'ac': 1, 'cg': 3, 'gc': 1, 'gt': 1, 'tc': 0, 'ag': 1}
Out-degree: {'ga': 2, 'ac': 1, 'cg': 3, 'gc': 1, 'gt': 0, 'tc': 1, 'ag': 1}
The graph has an Eulerian cycle.
```

2.5) Generating an Assembly

The generated graph has a Eulerian cycle, it has at most 2 odd degrees and all other are even degrees.

Flow: {TC, CG, GA, AG, AC, CG, GC, GT}

The assembled genome sequence would be: " TCGAGACGCGT ".

3) Difficulty Adjustment

In total assignment took around 14 hours to complete, initially to understand the concept took some time.

The implementation of De Bruijn Graph and the detection of Eulerian Cycles was bit confusing earlier and I am little bewildered whether to take k-1mer or k=2mer for plotting De Bruijn Graph.