Background:

Biology

DNA sequencing has been a paramount tool for understanding genomics. To find the DNA sequence of the genome (Fig.1A) in its entirety the first is to amplify the genome. This process is called whole genome amplification or WGA. WGA creates hundreds of identical copies of the given genome (Fig1.B). These genome copies can be ligated or cut randomly creating various length subsequences. The pool of subsequences from the cut genome copies is called a library (Fig. 1C). This library tagged using Sanger sequencing methods. A fluorescently tagged, dideoxynucleoside triphosphate for A, G, C, and T are introduced to DNA fragments mixture. The tagged nucleotides prevent DNA polymerase from adding another base pair to the stand thus quenching the experiment. The experiment labels all nucleotides in short DNA fragments by length. With known DNA sequence for the short subsequences, assembly of the DNA is possible by overlapping subsequences and slowly piecing together the sequence of the original genome (Fig.2).

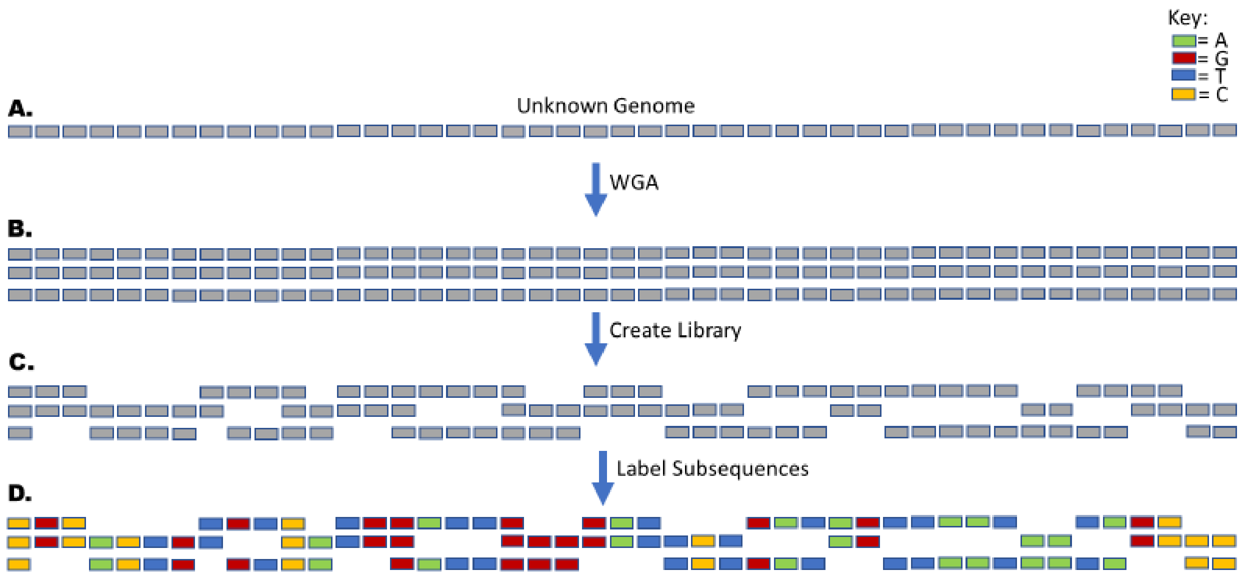


Figure 1. This is a crude depiction of DNA sequencing using shotgun techniques. A. Starting genome is unknown B. Using PCR techniques, whole genome amplification (WGA) of this genome is preformed C. Cuts at various lengths and positions are made to the WGA sample creating a library of short DNA fragments or subsequences D. These sequencing reads are identified through Sanger sequencing techniques. The DNA sequence of the genome can now be found by assembling the fragments

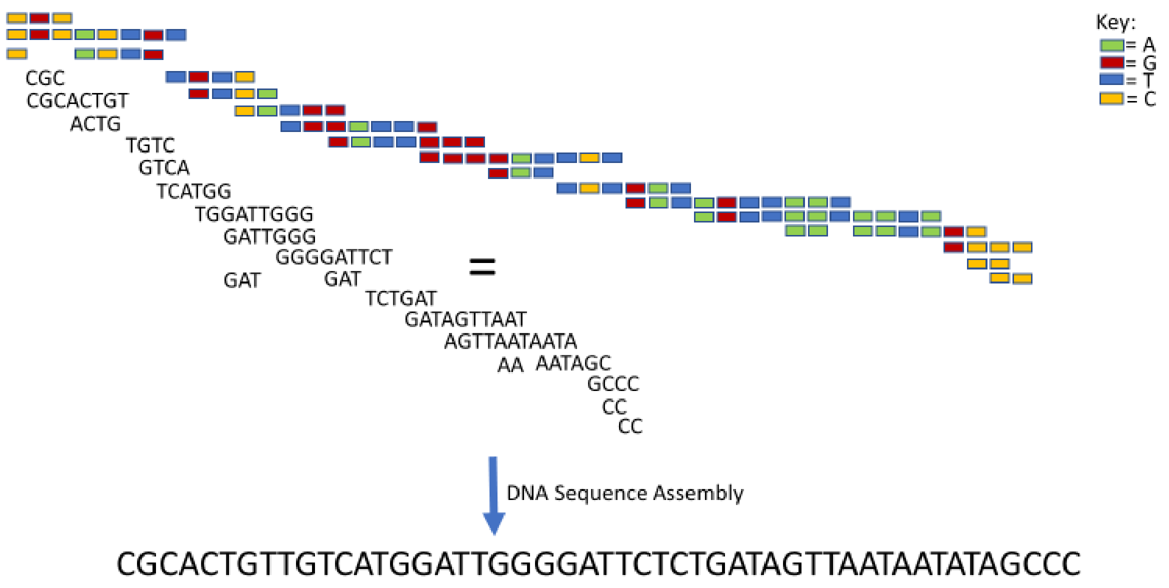


Figure 2. This is a crude depiction and overview of how DNA assembly is preformed. Known subsequences, of various length, will be overlapped onto one another forming one cohesive DNA sequence or genome.

Algorithm:

To accurately form a single genome sequence from a labeled subsequence library achieved by building a de Bruijn graph following Eulerian cycle path standards. The de Bruijn sequence approach will take two inputs, the subsequence and k-mer length. Edges of the graph consist of all possible k-mers in subsequence. This is found by a loop moving through subsequence with steps i. The i-th edge will equal i-th k-mer in string [1]. Once i > k-mer the loop returns all possible k-mers within the sequence (Fig.3). During this process vertices are labeled where the i-th (k-1)-mer in subsequence. These vertex redictions are stored for following loop. This loop will be preformed on every subsequence in library.

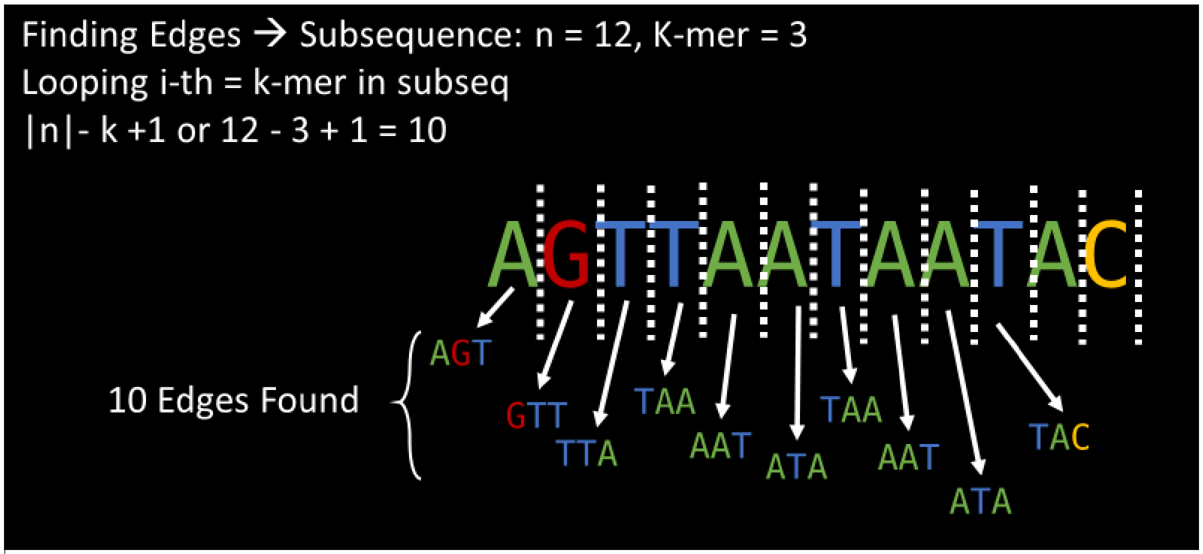

Finding Edges → Subsequence: n = 12, K-mer = 3
Looping i-th = k-mer in subseq
|n|- k +1 or 12 - 3 + 1 = 10
AGTTAATAATAC
10 Edges Found
AGT
GTT
TTA
TAA
AAT
ATA
TAA
AAT
ATA
TAC

Figure 4. The next step in the de Bruijn algorithm is sift through predicted vertices produced by the edge discovering loop. Predicted vertices values, i-th k-1, were labeled in the subsequence. By looping through these predictions, unique vertices can be identified and duplicate vertices are merged together so that associated edges remain. Here 20 vertices were predicted from the 10 edges. From those 20 predictions, 7 unique vertices were established. These will be used for the de Bruijn graph nodes.

A vertex in the de Bruijn graph represents the suffix and prefix of associated edge. All predicted vertex values, found and stored by pervious list, must be merged if duplicates exist. The path with input values, subsequence and vertices, will distinguish all vertices with unique nucleotide pair combinations (Fig. 4). Know a graph = (V, E) can be built because all edges have been accounted for in each subsequence as well as all unique vertex values. Vertex values will make up the nodes of the graph.

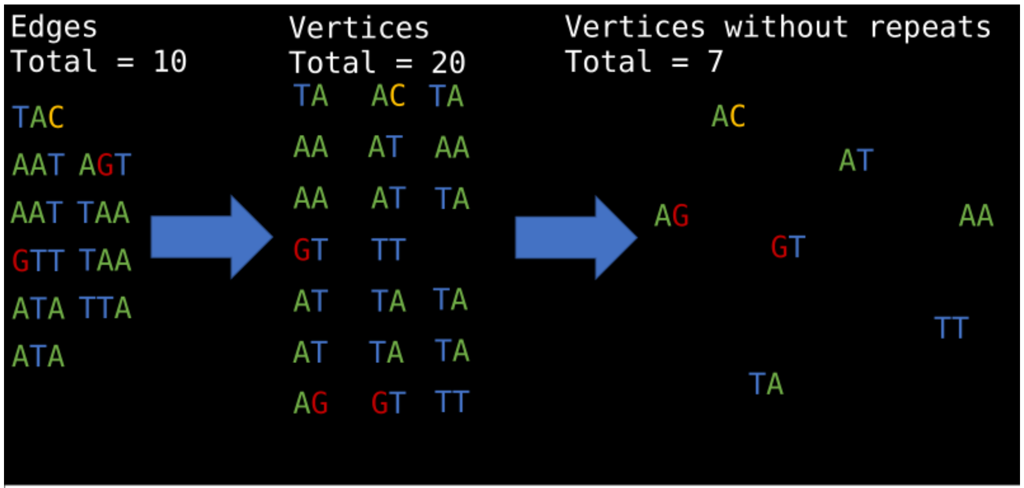


Figure 4. The next step in the de Bruijn algorithm is sift through predicted vertices produced by the edge discovering loop. Predicted vertices values, i-th k-1, were labeled in the subsequence. By looping through these predictions, unique vertices can be identified and duplicate vertices are merged together so that associated edges remain. Here 20 vertices were predicted from the 10 edges. From those 20 predictions, 7 unique vertices were established. These will be used for the de Bruijn graph nodes.

To propagate the final sequence, a directional graph must be built using unique vertices and all edges found in previous loops. The edges of the graph are directional in that they are either indegree or outdegree. Using Eulerian path standard, a cycle on the input graph where each edge is visited exactly once will enable assembly of each subsequence into a single cycle. where input is graph = (V, E) (Fig 5.).

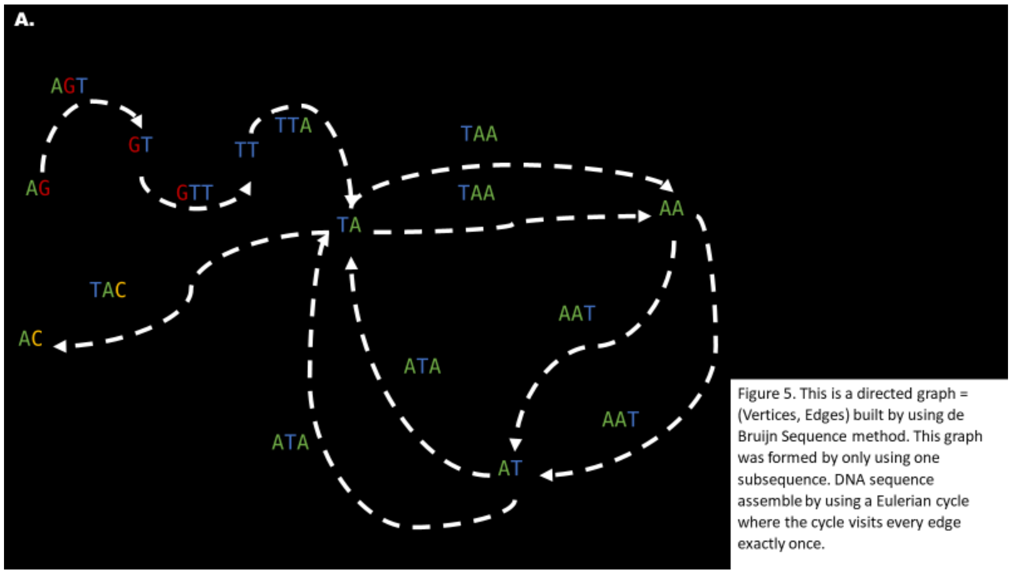
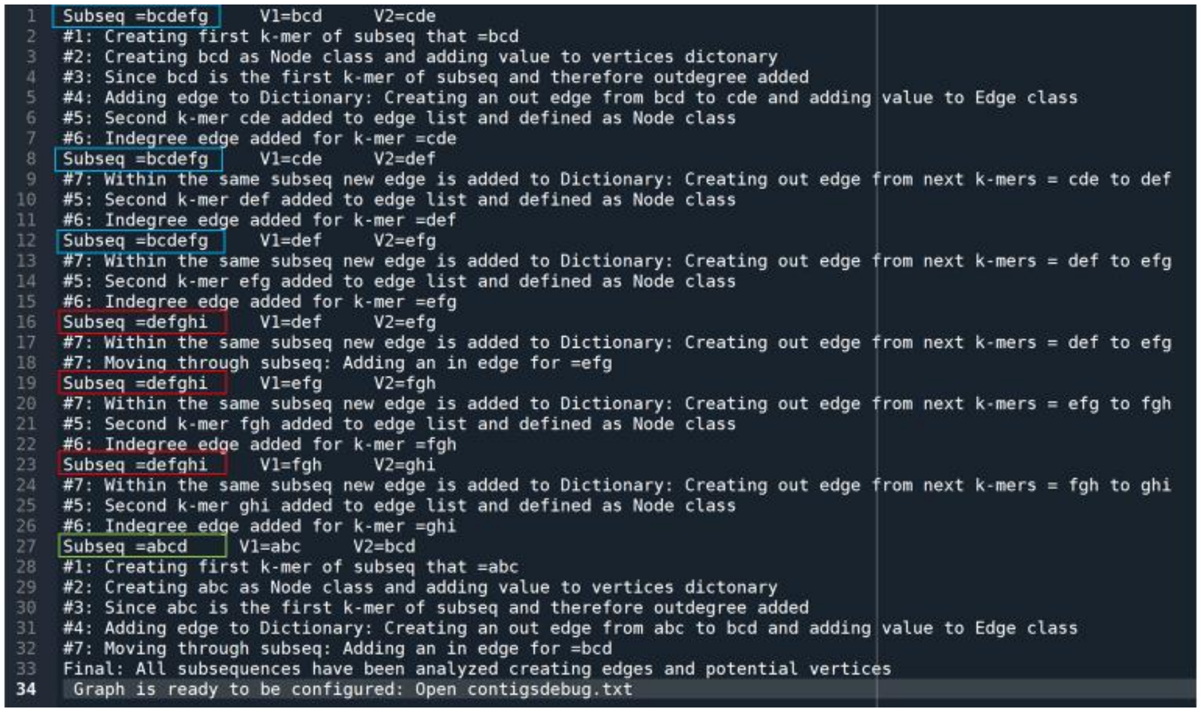


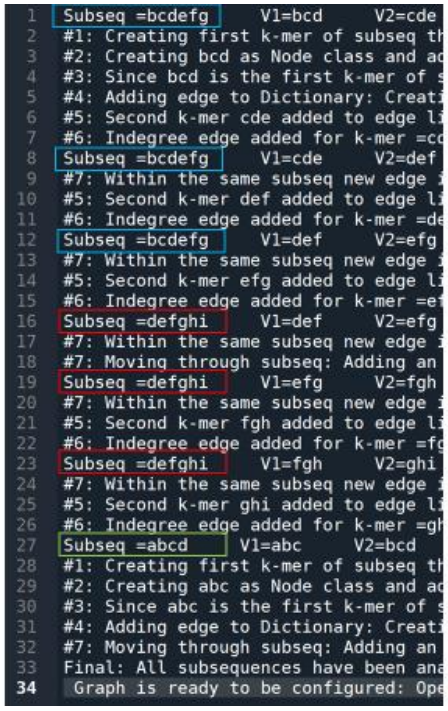
Figure 5. This is a directed graph = (Vertices, Edges) built by using de Bruijn Sequence method. This graph was formed by only using one subsequence. DNA sequence assemble by using a Eulerian cycle where the cycle visits every edge exactly once.

Implementation:

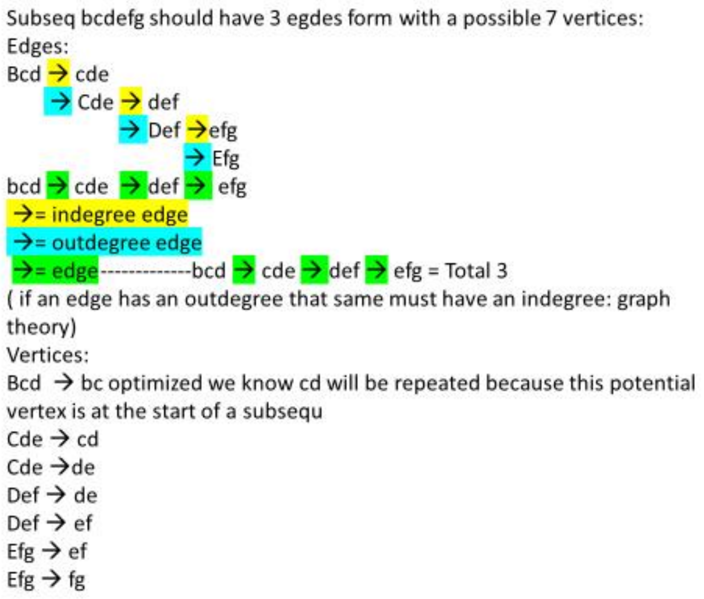
Using Python 3.7 on Windows 10 OS platform I was able to successfully integrate the de Bruijn algorithm into my model random sequence generator. Firstly, I created a basic pseudocode for my model: 1) Read all randomly chopped sequences 2) Construct graphs by looping through all subsequences. I started by debugging the mitbal's debruijn module on Github.com. I started off by breaking down what exactly the code was doing by using a simple test sequence set equal to 'abcdefghi'. Three function and two classes were defined. The functions included read_reads(fname) which opened up a file containing a library of subsequences. The function construct_graph(reads, k) where reads equals all subsequences in file and k equals k-mer length. This function creates a dictionary of edges and vertices as well as labeling Node class and Edge class values. The loop located in this function runs through each subsequence moving by k-mer increments. In the test run the subsequences were ['bcdefg', 'defghi', 'abcd']. In the construct_graph function bcdefg would considered a read of reads and broken down into all of it's possible k-mer which were set to equal 3. Two variables are defined inside of the loop v1 and v2. V1 represents the very first k-mer and v2 represents the next (v1= bcd, v2=cde). The very first k-mer of a subsequence fails the first if statement where v1 is first labeled as Node class and vertices dictionary. Node class also labels degree of the node, so when labeling a Node an outdegree and/or indegree is also labeled. The first k-mer will be given an outdegree of 1. This k-mer will then be labeled as Edge class and edge dictionary. Classes will be referred back to during the output_contig function. Moving on to the second k-mer(v2 =, 'cde'). V2 has not be manipulated at this point and thus has not be added to either dictionary causing it to fail first if statement. V2 is labeled Node class with indegree of 1 and finally added to the edge dictionary. Now that the first k-mers have been added the if statements should be True for the following k-mers where each will be added to Node and Edge class with appropriate indegrees and outdegrees assigned to them. This is all debugged in the file "fkmerdebug.txt". This debug file allows you to see how the function is walking through each subsequence. Shown is debugging file:



```
Subseq =bcdefg     V1=bcd     V2=cde
#1: Creating first k-mer of subseq that =bcd
#2: Creating bcd as Node class and adding value to vertices dictonary
#3: Since bcd is the first k-mer of subseq and therefore outdegree added
#4: Adding edge to Dictionary: Creating an out edge from bcd to cde and adding value to Edge class
#5: Second k-mer cde added to edge list and defined as Node class
#6: Indegree edge added for k-mer =cde
Subseq =bcdefg     V1=cde     V2=def
#7: Within the same subseq new edge is added to Dictionary: Creating out edge from next k-mers = cde to def
#5: Second k-mer def added to edge list and defined as Node class
#6: Indegree edge added for k-mer =def
Subseq =bcdefg     V1=def     V2=efg
#7: Within the same subseq new edge is added to Dictionary: Creating out edge from next k-mers = def to efg
#5: Second k-mer efg added to edge list and defined as Node class
#6: Indegree edge added for k-mer =efg
Subseq =defghi     V1=def     V2=efg
#7: Within the same subseq new edge is added to Dictionary: Creating out edge from next k-mers = def to efg
#7: Moving through subseq: Adding an in edge for =efg
Subseq =defghi     V1=efg     V2=fgh
#7: Within the same subseq new edge is added to Dictionary: Creating out edge from next k-mers = efg to fgh
#5: Second k-mer fgh added to edge list and defined as Node class
#6: Indegree edge added for k-mer =fgh
Subseq =defghi     V1=fgh     V2=ghi
#7: Within the same subseq new edge is added to Dictionary: Creating out edge from next k-mers = fgh to ghi
#5: Second k-mer ghi added to edge list and defined as Node class
#6: Indegree edge added for k-mer =ghi
Subseq =abcd     V1=abc     V2=bcd
#1: Creating first k-mer of subseq that =abc
#2: Creating abc as Node class and adding value to vertices dictonary
#3: Since abc is the first k-mer of subseq and therefore outdegree added
#4: Adding edge to Dictionary: Creating an out edge from abc to bcd and adding value to Edge class
#7: Moving through subseq: Adding an in edge for =bcd
Final: All subsequences have been analyzed creating edges and potential vertices
 Graph is ready to be configured: Open contigsdebug.txt
```



```
Subseq =bcdefg      V1=bcd     V2=cde
#1: Creating first k-mer of subseq th
#2: Creating bcd as Node class and ad
#3: Since bcd is the first k-mer of s
#4: Adding edge to Dictionary: Creati
#5: Second k-mer cde added to edge li
#6: Indegree edge added for k-mer =cc
Subseq =bcdefg      V1=cde     V2=def
#7: Within the same subseq new edge i
#5: Second k-mer def added to edge li
#6: Indegree edge added for k-mer =de
Subseq =bcdefg      V1=def     V2=efg
#7: Within the same subseq new edge i
#5: Second k-mer efg added to edge li
#6: Indegree edge added for k-mer =ef
Subseq =defghi      V1=def     V2=efg
#7: Within the same subseq new edge i
#7: Moving through subseq: Adding an
Subseq =defghi      V1=efg     V2=fgh
#7: Within the same subseq new edge i
#5: Second k-mer fgh added to edge li
#6: Indegree edge added for k-mer =fg
Subseq =defghi      V1=fgh     V2=ghi
#7: Within the same subseq new edge i
#5: Second k-mer ghi added to edge li
#6: Indegree edge added for k-mer =gh
Subseq =abcd      V1=abc     V2=bcd
#1: Creating first k-mer of subseq th
#2: Creating abc as Node class and ad
#3: Since abc is the first k-mer of s
#4: Adding edge to Dictionary: Creati
#7: Moving through subseq: Adding an
Final: All subsequences have been ana
 Graph is ready to be configured: Ope
```



Subseq bcdefg should have 3 egdes form with a possible 7 vertices:
Edges:
Bcd → cde
→ Cde → def
→ Def →efg
→ Efg
bcd → cde → def → efg
→= indegree edge
→= outdegree edge
→= edge-------------bcd → cde → def → efg = Total 3
( if an edge has an outdegree that same must have an indegree: graph theory)
Vertices:
Bcd → bc optimized we know cd will be repeated because this potential vertex is at the start of a subsequ
Cde → cd
Cde →de
Def → de
Def → ef
Efg → ef
Efg → fg

Once all k-mers are labeled as Edges, the next function output_contigs(g) takes the input g which equals the previous function result. This function is obviously optimized and skips steps where I would have used brute force to compile vertex information. The following assumptions are made when executing this function:

1. Assumes the Node with the least number of indegrees corresponds to the first k-mer of entire sequence
2. Assumes the subsequence will not split first k-mer sequence into separate nodes
3. Assumes that large k-mers will be used for DNA assembly
    a. Small k-mers in sequences with repeats length > k-mer length will compromise function

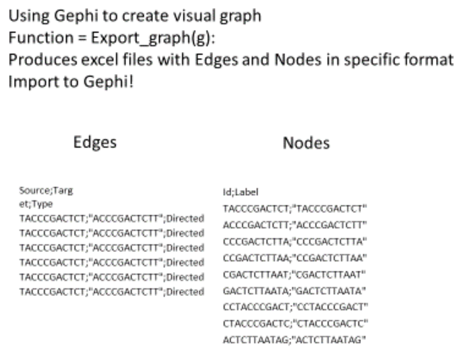
The function first obtains the most likely starting Node based off on the Node with the smallest indegree value. From the starting Node it verifies the presences of an edge, adds the node to 'contig' string which will eventually become the assembled sequence. The edge is deleted from the Edge class so to not repeat the edge at any point in the cycle. This falls inline with Eulerian cycle standards. One nucleotide is added to the contig string based off of outdegree from previous node and it's on indegree values. This function was debugged in file, "contigsdebug.txt", where each manipulation is translated into comprehensive text.

The final function incorporated in the debruijn.py file is print_graph(g) which takes input construct_graph(reads, k) and prints nodes and edges accordingly. After this function I created an additional export_graph(g) function that creates text files that can be imported to Gephi 0.9.2. Gephi gives a visualization of the graph structure and directionality of edges.

Verification and Analysis of the algorithm:

To verify this module I will simulate a DNA sequencing machine. First, randomly generate a genome, and store genome identity to validate DNA assemble defined by contig. Chop the genome at random sites on the genome of random size. This simulates WGA, library creation and labeling of subsequences. The program loops through the sequence 1000 times and randomly cuts sections of the sequence. This data is added to a text file which will be called upon in the debruijn.py program. Since the sequence was generated, we know the cut sequence identity which mimics Sanger sequencing ladder technique. My forward thoughts on improving the verification process would be to randomly inject errors into the chopped subsequence containing file and determine the error correction, identification, etc.

To simulate a sequence I created a function called random_seq(lenseq, lowsizedchop, highsizedchop). The input variables for this function are the length of the sequence (basepairs) you would like to generate (Default = 30); The lowest chop size of a subsequence and conversely the highest chop size of a subsequence. The sequence is built by randomly selecting nuc from nucleotide list and added to the sequence string 'seq'. Next an integer check was put into place so that invalid low or high size chops will be corrected. Next, I start chopping the sequence and adding subsequences to configseq file named 'SequenceFile.txt'. The front and tail sequences are manually added with lengths of 4 and -4 accordingly. This is to emphasize that the start and end of the genome are successfully included in the subsequence library. A for loop where a random location is defined and a random chop size is defined. A random chop is made following the give chop size parameters and added to 'SequenceFile.txt". Originally, I had set the k-mer size equal to 3 but this was entirely too small and would create edges that looped over and over again at the same subsequence. The small k-mer size revealed possible errors that would occur when assembling a sequence with runs of identical nucleotides. For example: input genome file that should produce DNA sequence GTGAAAGGGG actually would produce GTGAAAAGGTGG. This could be due to the assumption listed above. This de Bruijn module, does not clearly define vertices as the prefix and suffixes as they should be which allows for errors such as these to occur. Run random_seq will generate a sequence which will then move into the debruijn.py program where it will be read, labeled/constructed and assembled. The random sequence should be printed first and the assembled sequence will be below it. Are these sequences identical? If not run work_1.txt or work_2.txt in place of SequenceFile.txt. Shown to the left is the graph I was able to produce in using Gephi. The input needed to produce this graph is the output of export_graph function. Here the k-mers are equal to 11 and graph created is directional.



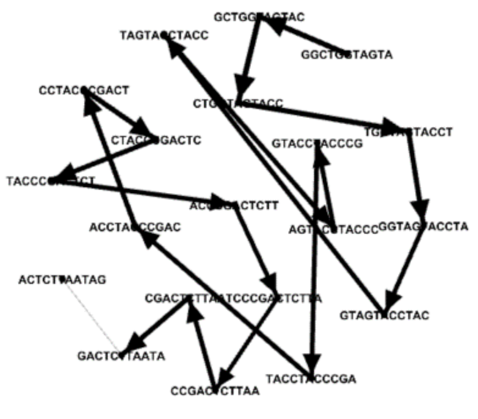
Using Gephi to create visual graph
Function = Export_graph(g):
Produces excel files with Edges and Nodes in specific format
Import to Gephi!

Edges

```
Source;Targ
et;Type
TACCCGACTCT;"ACCCGACTCTT";Directed
TACCCGACTCT;"ACCCGACTCTT";Directed
TACCCGACTCT;"ACCCGACTCTT";Directed
TACCCGACTCT;"ACCCGACTCTT";Directed
TACCCGACTCT;"ACCCGACTCTT";Directed
TACCCGACTCT;"ACCCGACTCTT";Directed
```

Nodes

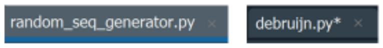
```
Id;Label
TACCCGACTCT;"TACCCGACTCT"
ACCCGACTCTT;"ACCCGACTCTT"
CCCGACTCTTA;"CCCGACTCTTA"
CCGACTCTTAA;"CCGACTCTTAA"
CGACTCTTAAT;"CGACTCTTAAT"
GACTCTTAATA;"GACTCTTAATA"
CCTACCCGACT;"CCTACCCGACT"
CTACCCGACTC;"CTACCCGACTC"
ACTCTTAATAG;"ACTCTTAATAG"
```



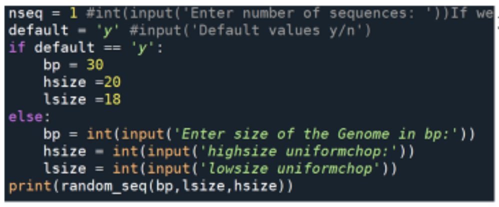
How to:

1. Open random_seq_generator.py and debruijn
2. I will have the program set to the default I created
    1. If you would like to manipulate the sequence to test the algorithm simple switch default = 'y' to default = 'n'
3. Set fname to 'SequenceFile.txt' to generate new sequence or you can switch this out for sequences seen to successfully assemble (work_1.txt or work_2.txt)
4. 'SequenceFile.txt' should be generated within the same folder as the py files
5. Debugging files will also appear: contigsdebug.txt and fkmerdebug.txt
6. Garph.csv, Nodegraph.csv and Edgegraph.csv are used for Gephi input
7. Run: First line = generated sequence; Second line = Assembled sequence

1

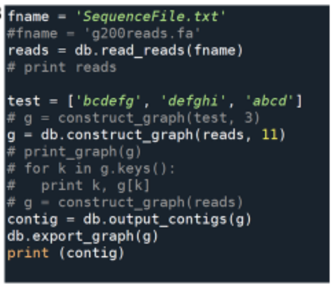


2



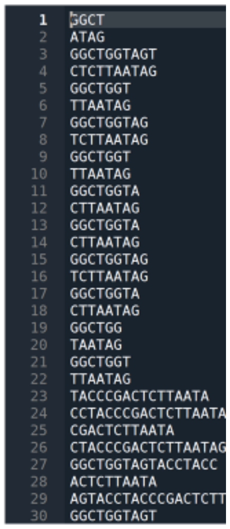
```
nseq = 1 #int(input('Enter number of sequences: '))If we
default = 'y' #input('Default values y/n')
if default == 'y':
    bp = 30
    hsize =20
    lsize =18
else:
    bp = int(input('Enter size of the Genome in bp:'))
    hsize = int(input('highsize uniformchop:'))
    lsize = int(input('lowsize uniformchop'))
print(random_seq(bp,lsize,hsize))
```

3



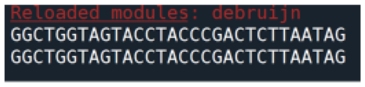
```
fname = 'SequenceFile.txt'
#fname = 'g200reads.fa'
reads = db.read_reads(fname)
# print reads

test = ['bcdefg', 'defghi', 'abcd']
# g = construct_graph(test, 3)
g = db.construct_graph(reads, 11)
# print_graph(g)
# for k in g.keys():
#   print k, g[k]
# g = construct_graph(reads)
contig = db.output_contigs(g)
db.export_graph(g)
print (contig)
```

4



```
GGCT
ATAG
GGCTGGTAGT
CTCTTAATAG
GGCTGGT
TTAATAG
GGCTGGTAG
TCTTAATAG
GGCTGGT
TTAATAG
GGCTGGTA
CTTAATAG
GGCTGGTA
CTTAATAG
GGCTGGTAG
TCTTAATAG
GGCTGGTA
CTTAATAG
GGCTGG
TAATAG
GGCTGGT
TTAATAG
TACCCGACTCTTAATA
CCTACCCGACTCTTAATA
CGACTCTTAATA
CTACCCGACTCTTAATAG
GGCTGGTAGTACCTACC
ACTCTTAATA
AGTACCTACCCGACTCTT
GGCTGGTAGT
```

7



```
Reloaded modules: debruijn
GGCTGGTAGTACCTACCCGACTCTTAATAG
GGCTGGTAGTACCTACCCGACTCTTAATAG
```

References:

1. Jones, Neil C., and Pavel Pevzner. *Introduction to Bioinformatics Algorithms*. MIT Press, 2014.
2. Mitbal. "Mitbal/Py-Debruijn." *GitHub*, github.com/mitbal/py-debruijn.git.
3. Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W. Shen, Mark Chaisson, Pavel A. Pevzner. Assembly of long reads using de Bruijn graphs,Proceedings of the National Academy of Sciences Dec 2016, 113 (52) E8396-E8405; DOI: 10.1073/pnas.1604560113

4. Zinoviev, Dmitry. *Complex Network Analysis in Python: Recognize - Construct - Visualize - Analyze - Interpret*. The Pragmatic Bookshelf, 2018.