Zhenyu Wu

CSE-308

Prof. Brian Chen

27th February 2023

**Section 1**: Before I dive into the details of project implementation, I would like to go over the general steps that I followed to build my code. All the implementations are done within the assemble.py file in the Implementation folder. To read the input file, I imported the sys library which can accept input from the command line which allows users to choose an input file on their own and pass it to the assembler. Then, It will parse the raw input fasta file into a list of reads. After that, I break each read into two parts with equal length: prefix and suffix, encode the prefix and suffix separately, record the outdegree and indegree of each node and insert them into a hash table with 3 layers. After setting up the hash table, I applied the method mentioned in class–The Eulerian Path Assembly, which first selects a node with larger outdegree than indegree and goes through possible suffixes through edges until there exists no edge that can lead us to find another node, and I will mark used edges to prevent reuse the same edges. Repeating the Eulerian Path Assembly until there are no nodes in the hash table with higher outdegree than indegree will give us a hash table that contains only cycles. Then, I perform the walk through assembly method once again to assemble the cycle. As we gather cycles and paths, we will try to compare the first node of a cycle with every node of path to find the overlapping region to join the cycle into the path, which means each cycle will need to go through the whole list of paths to find a possible joint. When paths and cycles join successfully, decoding the contigs from numbers back to the original form is needed to perform the overlapping test. After we get back the original form, since many of them are just duplicates or shorter versions of other longer contigs, I perform a cleaning process on the list of contigs to clean all the duplicates and shorter incomplete ones, which will leave contigs that are longer and unique. One last step is to join the contigs by finding their overlapping region at head or tail, which is simply join overlapping contigs head to tail, and it will give us the final output–Contigs that are unique to each other.

Firstly, I will briefly introduce how I implement the parsing, encoding, decoding, and finding degrees. For parsing, I just simply use the “read\_line” method to read a fasta file and skip lines with naming and reads’ IDs. The encoding and decoding processes are similar but in the reverse process, which just follow the methodology that V = n4^0 + n4^1 + n4^2 + …. + n4^(n-1). In the above formula, n represents the specific number that we assigned to each nucleotide where A = 0, C = 1, G = 2, T = 3. By encoding prefixes and suffixes in this way, we can get a hash number for them which can be used to implement a hash table. To translate back from a hash number to a prefix and suffix, we divide the hash number by 4^i each time, the floor of V/4^i will give us the nucleotide at i position. Then we repeat this process from i = n-1 to i = 0. For finding indegree and outdegree, since I have stored prefixes and suffixes into two separate lists, finding degree is just a simple process of recording the occurrence of each prefix and suffix in the prefix list and suffix list. Prefix list will give us an outdegree and suffix list will provide us indegree. Above functions have complexity O(n) since they need to go through each element in the list, but they are definitely worth implementing, especially finding a degree function. Since I store degrees in a dictionary something like {prefix or suffix: degree} which means it will give us constant searching and operating time when we perform the Eulerian Path Assembly which is more time consuming. Such optimization is needed which prevents the situation that we need to nest another loop into the outer loop to find the indegree and outdegree.

The hash table is actually the simplest part of the implementation. It is a three-layer dictionary where the first level dictionary stores each prefix as key and second level dictionaries as its value. For the second-level dictionaries, they store suffixes that link to the same prefix as key and the number of edges between shared prefix and key as value. For example, {prefix\_1: {suffix\_1:1, suffix\_2:3}, prefix\_2:{suffix\_3:1}, …}. The function that builds the hash table benefits from the separate list of prefixes and suffixes. Since each element in both lists with the same index are the prefix and suffix that build a read which means they have the same length (the total number of reads we received). Therefore we just need to increment the iterator from 0 to the total number of reads we received to go through two lists simultaneously and insert all nodes into the table, which gives us a complexity as O(n) instead of O(n+n).

One of the most important algorithms is the Eulerian Path Assembly to generate lists of paths and cycles. To perform such algorithms, the first step is to start with a prefix that has greater outdegree than indegree. In my code, there is an outer for loop, looping through every key(prefix) in the first level of the hash table. Inside the outer for loop, there is another inner while true loop to keep generating a path for the specific key that the for loop is working on. Inside the inner while loop, it will first calculate the outdegree and indegree of this prefix. If outdegree <= indegree, while loop will break and go to the next prefix in the first level. In contrast, if the outdegree > indegree, we will pass the table, prefix, outdegree and indegree lists to another function to generate the paths(called read\_path()). Inside of the read\_path() function, it will create an empty list for the current path, add the prefix we passed as the head of contig, and then use a while loop to go through every suffix that can be visited through edges until we meet a node (or suffix) that has no edge to another nodes. During the iteration process, the function keeps track of the outdegree, indegree and used edges to ensure that we will not reuse the same edge, and every node’s outdegree and indegree will be correct. Then read\_path() will return the one path per running, and I append each output path to the list of paths. After the outer for loop terminates, which means we only have nodes with outdegree = indegree in the table, then I call the read\_path() again within another for loop to go through unused nodes to generate a list of cycles. Eventually, it will produce two lists: a list of paths and a list of cycles. This algorithm has a complexity of O(n\*k\*c) where k is the case when a node has larger outdegree than indegree, and c represents the number of nodes that a specific prefix can reach to.

Once we have lists of path and cycle, the code will perform a function to join the path and cycle together. Firstly, it will go through a for loop to search for every possible path that can bind with cycles, add them into a target path list. Then, because we haven’t cleaned the identical path, it will call another function to clean the duplicates in the target path list. After cleaning, I just bind each cycle to one of the possible paths in the target path list. Finally, replace the original unbinded paths with our paths that are already binded with cycles.

After binding path and cycle, since there are still many identical paths or paths that are covered by other longer paths. It is time to delete shorter and identical ones. This function reads through every single path in the paths list, checking if it is covered in any path in the rest of the path list and in the output list. If not, add to the output list, which makes every path in the output list unique. This function has a complexity of O((n+n)^2) since it will loop through all elements in the two lists in every iteration. Therefore, the general performance of this algorithm is not good, especially for long input list(eg. Herpes has 99311 elements in the list), which needs nearly half a minute to finish. However, the reason why I keep this function is to speed up the process of merging contigs with overlapping regions, because it will shrink the size of the input array to nearly 1/50 of the original one (eg. Herpes shrinks from 99311 to 2839), which is another optimization I think is worthy to implement.

Eventually, we will merge contigs with overlaps together to generate longer contigs. This involves two functions. The first one is to detect the overlapping region between two input strings. Most of the code is done in the while loop, in this loop we will repeatedly compare str\_1’s first i character with str\_2’s last i character and str\_2’s first i character with str\_1’s last i character, if there exists a overlapping region that has a minimum length of 15(i >=15), we will record it as possible overlap. Then when I exceed one of the string lengths, it will break the while loop, then I will return the longest overlapping region in the list. This function has O(n) complexity. The second one is to compare and attach contigs together. It is started with a for loop: for length of list != 0. Inside the for loop, there is another nested while loop. For each iteration of the outer for loop, it picks the first element as pivot, then enters the while loop, and starts the next iteration when the while loop terminates. Inside the while loop, it will firstly set old\_pivot = pivot which is used to test if current pivot merges other contigs after innermost iteration. Then, throw the pivot into the innermost for loop which loop through 0 to the length of the list, compare pivot with each element by the detection function I mentioned above. If overlap exists, merge them, and mark the elements merged with pivot as used. After this innermost iteration, compare old\_pivot with pivot, if pivot is the same as old\_pivot which means we can not extend the current pivot anymore and break the while loop. Before we break the while loop, we also clean up all elements that are marked as used, which will make the list contain only unused elements. Then the next iteration of outer for loop will start with the first unused element. This complicated function is the most time consuming algorithm with O(n(n-i)^2) as complexity where i represent the number of elements we already merged, which is the reason I perform the cleaning function before merging. If I pass in the raw list of paths with duplicates, it will take tens of minutes to run, but after cleaning, it will only take nearly 20 seconds for Herpes.

In addition, I also found a fact that the result of merging overlapping contigs might be slightly different when I pass in the same contig list but elements are in different order. This is because the algorithm I used to bind contigs together will pick a contig and try to bind with the rest contigs in a for loop which will iterate through each element one by one. Therefore, it may cause the problem I mentioned above. For example, contig A can bind with B and C. However, through iteration, it will bind with B first to form a longer contig D, but D may have a chance that can not bind with C. Therefore, the order of elements may influence the result. To deal with this problem, I wrote another modified version of merging function, which will add every overlapping region we found to a list in each iteration(eg. for contig B, C with contig A) instead of merging them immediately. Then, each time we only merge the one with the longest overlapping region to the current contig we are working with(eg.A). This modified version solves this problem which produces the same result without the influence of order. However, the downside is that it dramatically increases the complexity of the merging process from O(n(n-i)^2) to O(n(n-1)^2). The difference of result of both versions are within 1-2 contigs. The one that my code is using is the original one, but I also included the modified one in my source code.

In conclusion, my implementation’s weakness is its performance, because most of the functions and algorithms have complexity of O(n^2) or even O(n^3), which are really slow when dealing with a massive amount of input. In contrast, all the functions and algorithms can work correctly, each one of them perform their functionality in the right way and generate correct results in return value, which are tested manually. In terms of final result, it can assemble the contigs that are identical to the result generated by SSAKE for the Yeast Reads, but it will generate 68 contigs for herpes reads compared to 19 contigs generated by SSAKE.

**Section 2**: During the implementation process, I found it is hard to sequence the cycle and path. Normally, a cycle contains nodes that have indegree = outdegree, and it starts and ends with the same nodes. The way I join the cycle and path is just go through every path in the path list,.If there is one path containing a node that is the same as the head and tail nodes of cycle, I will insert the cycle into the path at the position of the shared node. However, this method has two problems that may affect the final output of sequencing. The first one is What if there exist more than one paths that are not duplicates to each other having the same shared node with cycle, which means cycle can be inserted into multiple different paths, and different paths will generate different final outputs. The second problem is that even if there is only one path that has the shared nodes with cycles, this path may contain multiple shared nodes at different positions, which means we can insert cycles in different positions and produce different output. The current method I used is just pick the first path that has the shared node with the cycle, and insert the cycle into the first position where the shared node appears. This method will restrict the number of possible outputs to only one output, which reduces the ambiguity to zero. However, if inserting cycles to different paths in different positions may produce longer contigs, our output will be inaccurate. This might be the reason why I receive perfect output from Yeast reads but not from Herpes reads, because Yeast doesn’t generate any cycle. One possible method to avoid such issues might be to try to insert cycles into all target paths and target positions, and record every possible outcome of joining cycles to paths. Somehow, this method raised another problem: performance, especially when we try to put it into the last function: merging overlapping contigs which has an complexity of O(n^3). If we have k possible paths to insert to and each has c possible positions to insert to, we will end up with k\*c more paths needed to be processed. In addition, after we merge the overlapping contigs, we also need to find contigs that contain those possible path & cycle combinations, determine which one is the longest contig and keep the longest contig. Eventually, remove the cycle from all other contigs. In summary, this method might be able to address the problem of restricting the ambiguity, but it is inefficient. Like I mentioned, for each cycle, it will add k\*c more elements to the list. If we have i cycles, it will add i\*k\*c elements to the list which largely increase the processing time of merging overlapping contigs. Unfortunately, I didn’t have enough time to implement this function, but I have tried to join every possible path to cycles by hand to test the result for Herpes. The difference of the number of contigs generated is still within 1-3 contigs.

**Section 3**: Generally speaking, gene expression profiling is to look into the cell itself and try to find out if genes are functional, and what kind of thing the genes can produce, the study of the way genes are transcribed to synthesize functional gene products — functional RNA species or protein products. Gene expression profiling measures which specific genes from the entire genome are being expressed in particular cells at any given time. It often measures the mRNA level, showing the pattern of genes expressed by a cell at the transcription level. As the result of gene expression profiling, we can know how a cell is functioning at a specific time. This is because cell gene expression is influenced by external and internal stimuli, including whether the cell is dividing, what factors are present in the cell's environment, the signals it is receiving from other cells, and even the time of day. For example, certain genes are known to be involved in cell division; if these genes are active in a cell, you can tell the cell is undergoing division, or whether a cell is differentiated. In terms of the benefits of applying DNA arrays in gene expression profiling, the most straightforward one is that DNA arrays allow the analysis of thousands of genes simultaneously, making it possible to study gene expression patterns on a genome scale. DNA arrays also allow scientists to determine how much of a particular gene is being expressed in a given sample by quantitative measurements.

SNP detection is just detection of SNP. SNP, Single nucleotide polymorphisms, is a genetic variation among people, which affects only one of the basic building blocks—adenine (A), guanine (G), thymine (T), or cytosine (C)—in a segment of a DNA molecule. For example, the replacement of an A for a T in the nucleotide sequence CGGTAC which produces a sequence of CGGTTC is an example of SNP. SNPs act as chromosomal tags to specific regions of DNA, and these regions can be scanned for variations that can have important implications for an individual's health and disease risk. DNA arrays can speed up the process of SNP detection by allowing the detection of thousands of SNPs simultaneously in a single experiment. Additionally, DNA arrays can be used for genome-wide association studies (GWAS) to identify SNPs that are associated with complex diseases such as diabetes, cancer, and heart disease and to determine how these polymorphisms are distributed across different populations.. GWAS typically involve comparing the genotypes of thousands of SNPs between cases (individuals with the disease) and controls (individuals without the disease), and identifying SNPs that are significantly more common in cases than in controls.

Reference:

Nature Publishing Group. (n.d.). Nature news. Retrieved March 1, 2023, from <https://www.nature.com/scitable/topicpage/genetic-variation-and-disease-gwas-682/#:~:text=GWAS%20seek%20to%20identify%20the,are%20distributed%20across%20different%20populations>.

Introduction to gene expression profiling. Thermo Fisher Scientific - US. (n.d.). Retrieved March1,2023,from https://www.thermofisher.com/us/en/home/life-science/pcr/real-time-pcr/real-time-pcr-learning-center/gene-expression-analysis-real-time-pcr-information/introduction-gene-expression-profiling.html

Introduction to gene expression profiling. Thermo Fisher Scientific - US. (n.d.). Retrieved March1,2023,from https://www.thermofisher.com/us/en/home/life-science/pcr/real-time-pcr/real-time-pcr-learning-center/gene-expression-analysis-real-time-pcr-information/introduction-gene-expression-profiling.html