David Armas

Cynthia Tovar

Ryan Schwarzkopf

March 28, 2023

**Genome Sequence Assembly Algorithm Analysis**

**Abstract**

Genome sequencing is a difficult task to handle. In this analysis computational methods are developed to assemble a sequence from a list of kmers. The process starts with a list of kmers and connects every kmer via prefix and suffix to make a deBruijn graph. From the graph a eulerian path can be assembled, and then the original sequence can be found. Different methods were implemented and tested for efficiency. Methods tested were by string comparison, with a hash table, and with a hash table and double ended queue. String comparison was the most efficient, while hash table with deque was the second most efficient and hash table without deque was the least efficient.

**Introduction**

Genome sequencing is a vital tool in bioinformatics. DNA can be a long sequence and it is difficult to replicate it from beginning to end. To work around this issue, shorter fragments called reads are assembled to obtain a certain part of a DNA sequence. Genome sequencing is achieved by utilizing k-mers. K-mers are small fragments of a DNA sequence, they can be 3 nucleotides short. This helps to divide this large task into small building blocks. (Compeau & Pevzner, 2018)

By dividing the genome into k-mers, a graph can be created to visualize possible genome assembly sequences. A deBruijn graph visualizes all possible paths between a given set of k-mers. If a deBruijn graph is created, there are still multiple paths that can be chosen. To determine the best path, a Eulerican cycle is found within the deBruijn graph. A Eulerian Cycle is a path within the graph that touches each edge exactly once. If a Eulerian Cycle exists for the given k-mers, that is the resulting genome sequence. (Compeau & Pevzner, 2018)

**Methods**

To assist with the processing, a skeleton code was provided in C++. Attempts were made to utilize this code and program all of the sequencing tasks in C++. This proved to be a huge learning curve. After many attempts, the sequencing tasks were created using Python. This change in programming language required the C++ skeleton code to be converted to Python.

*Task 1 - Create a hash function to match the prefix and suffix of a given k-mer*

Task 1 is completed by the function “genomePath”. This function 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. If “append\_last” is true, the function adds the remaining characters of the last k-mer to the end of the resulting string. This is done by using the string slicing notation [1:], which extracts all characters in the k-mer except for the first character. The function iterates through each k-mer in the given kmers, extracts the first character of each and concatenates the characters to form a new string which represents the genome.

*Task 2 - Construct a deBruijn graph from a group of k-mers*

This task is completed by the function “debruijn\_graph\_from\_kmers”. A list of DNA sequences is provided as input. A list of kmers is created which contains all the suffixes of each input sequence. A helper function “suffix\_composition” performs this task. The “suffix\_composition” generates all of the suffixes of a given length “k” from a given pattern string. The resulting kmers list is created without duplicates for better processing. This list of kmers is used to create a dictionary to represent the deBruijn graph. The function iterates over each input sequence kmer and adds an entry to the dictionary. The key of the entry is the prefix of the kmer and the value is the suffix of the kmer. The resulting dictionary is returned to represent the deBruijn graph.

For the construction of the deBruijn graph, the initial code was using a hash table with adjacency lists. Using these lists, cause the graph creation to be quadratic instead of linear. In order to fix this, a hash table was created with adjacency deques. Using the deque will allow the runtime to change from quadratic to linear. This structure will improve the eulPath method runtime as well.

*Task 3 - Implement a Eulerian Cycle finding algorithm*

The function eulPath completes task 3. This function takes a dictionary representation of the deBruijn graph as input and returns the Eulerian cycle 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. Then the last node is added to the path and removed from the stack. To perform this, an empty stack and a dictionary called “balanced\_count” is created by a helper function “balanceCount”. A node is added to the stack, which is the node with an imbalance of -1, meaning the node has one more outgoing edge than incoming edge. By starting with this node, the existence of a Eulerian Path is guaranteed. From this node, the code iterates through the given dictionary. It starts with the node at the tope of the stack and moves to the first adjacent node. If an adjacent node is found, it is added to the stack and removed from the given dictionary. If an adjacent node is not found, the top node is added to the stack and removed from the dictionary. No adjacent nodes indicates that there are no unexplored edges. The path list is reversed using the slide notation [::-1] and is returned. (*Additional information about the helper functions are provided in the supplemental Files - Bioinform\_Methods\_P2)*

*Task 4 - C++ to Python Conversion*

The previous tasks were implemented in python, however test cases needed to be implemented according to the specifications in the assignment. There were two options: the first option was to translate the python code to C++ and test in C++. The other option was to convert all of the testing methods from the C++ skeleton to python and test the methods in python. There was only one group member who had some familiarity with C++, but all of our group had some familiarity with python. It was decided to translate the C++ skeleton to python in order that all group members could do testing and debugging. Also there is a lot of overhead in python methods, so translating a simple dict or deque needs actual implementation and not just translation. To translate the C++ skeleton to python was fairly trivial in comparison. The biggest logic change was that the structs: DiGraph and Node were no longer needed so some code needed to be manipulated to account for that. A deque was also used to construct a Eulerian Path so the build sequence needed to be built from scratch to support a deque. Build\_sequence joins every node in the deque, and then concatenates every other letter in that string to a new string to get the actual build sequence. The deBruijnByStringComp and has\_Eularian\_Path were harder to translate, but the co. The test.cpp, k-assembler.cpp, and sequence.cpp could essentially be translated straight over to python without any major logic changes.

*Task 5 - Integration between converted code and task’s code*

The C++ to Python converted code was running functions in a different order than the newly created code required. One of the major changes was the use of the has\_Eulerian\_path function. The converted function was checking for node balancing. During the creation of tasks 1-3, node balancing was taken into consideration.

After the graph creation, balancedCount was called to determine the balance count of each node. The results of this function were used to verify if a Eulerian path exists. The function find\_Eulerian\_path was created to check these results. If a node within the graph has a balance of -1, then a Eulerian path exists and the function returns True. If there are no nodes with a balance of -1, the function returns False indicating that no Eulerian path exists.

An issue with example 0 “aaaaaaaaaaa” was causing balancedCount to indicate that no Eulerian path existed. This was because the graph contained a single node, so it did not have a balance of -1. This was resolved by setting the balance to -1 when a single node exists. This allowed has\_Eulerian\_path to indicate that a Eulerian path exists and all normal processing continued as expected.

After find\_Eulerian\_path is called, if it returns True, eulPath is called to generate the Eulerian path. If it returns False, the rest of the sequence processing is skipped and the code continues to the next sequence. Once the Eulerian path is determined, genomePath is called to generate the resulting genome.

**Results - *Full output is available in supplemental files (output.txt)***

1. Testing k-assembler by k-mer pairwise comparison

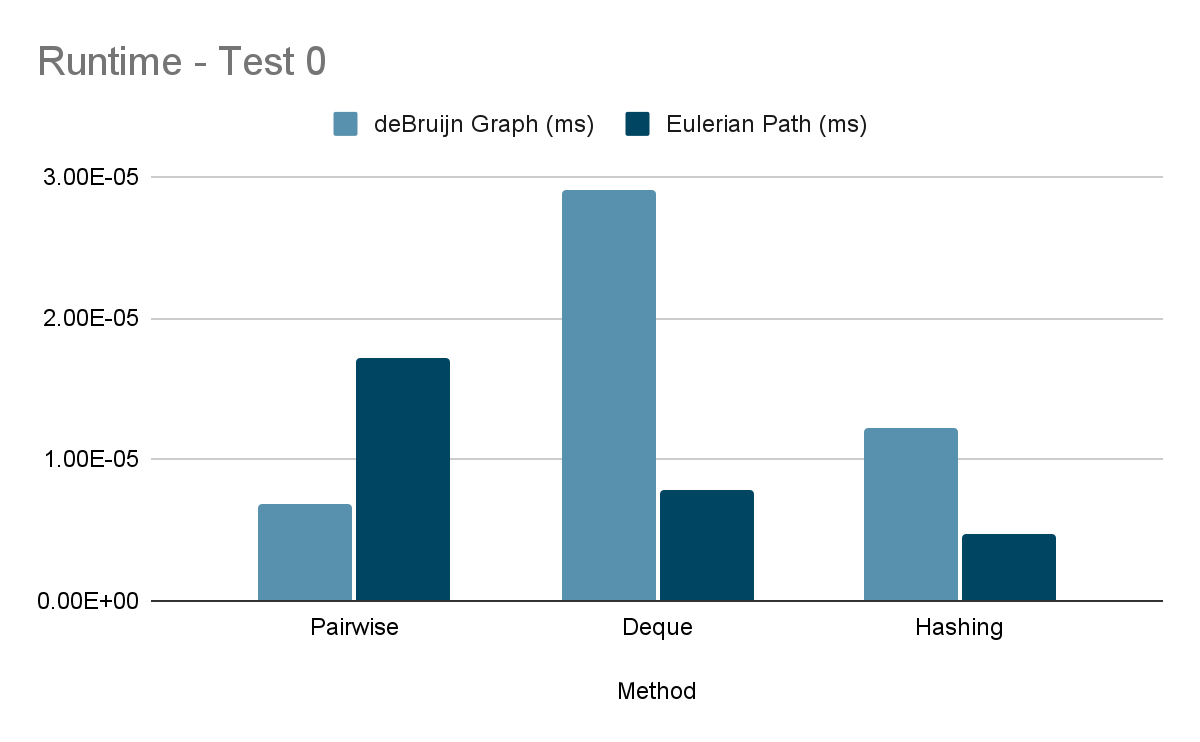
| DNA Sequence | deBruijn Graph (ms) | Eulerian Path (ms) | Assembled Sequence |
| --- | --- | --- | --- |
| Test 0: “aaaaaaaaaaa” | 6.9141387939453125e-06 | 1.71661376953125e-05 | Passed |
| Test 1: “agcagctcagc” | 5.0067901611328125e-06 | 8.106231689453125e-06 | Passed |
| Test 2: “agcagctcagg” | 6.198883056640625e-06 | 7.152557373046875e-06 | Passed |
| Test 3: random\_DNA\_sequence(10000, 20000) | 0.00964498519897461 | 0.007018089294433594 | Passed |

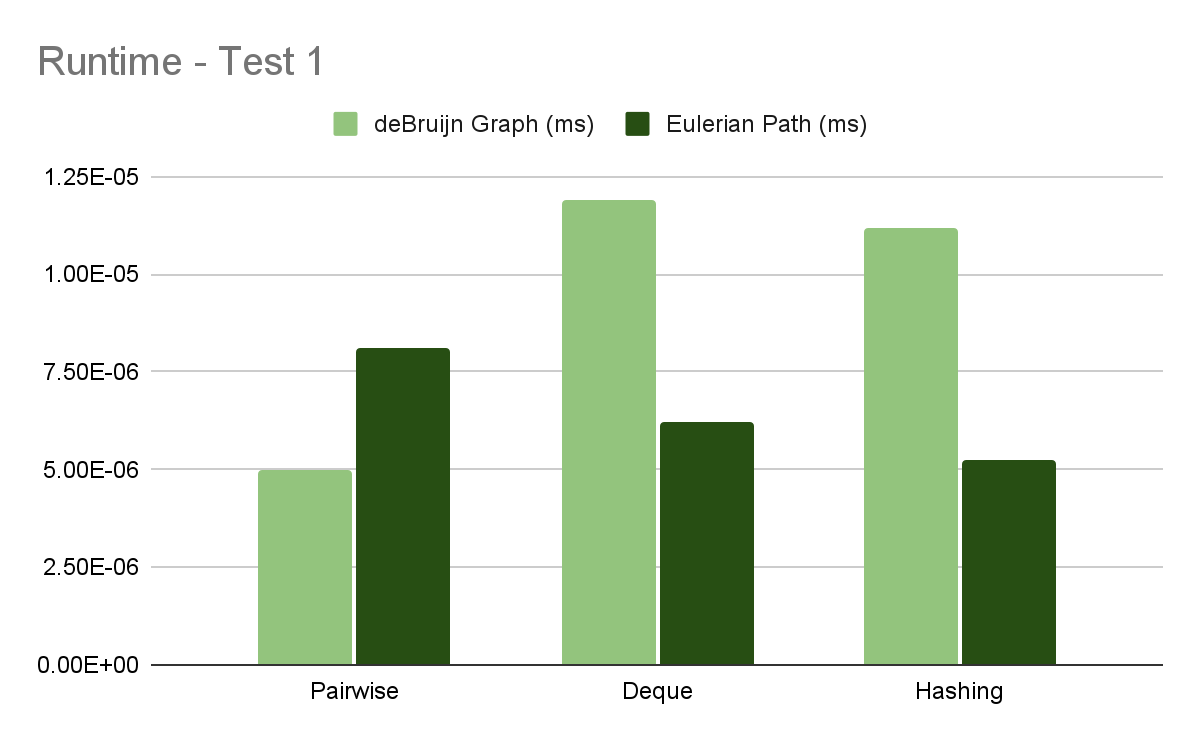
1. Testing k-assembler by k-mer hashing with deque

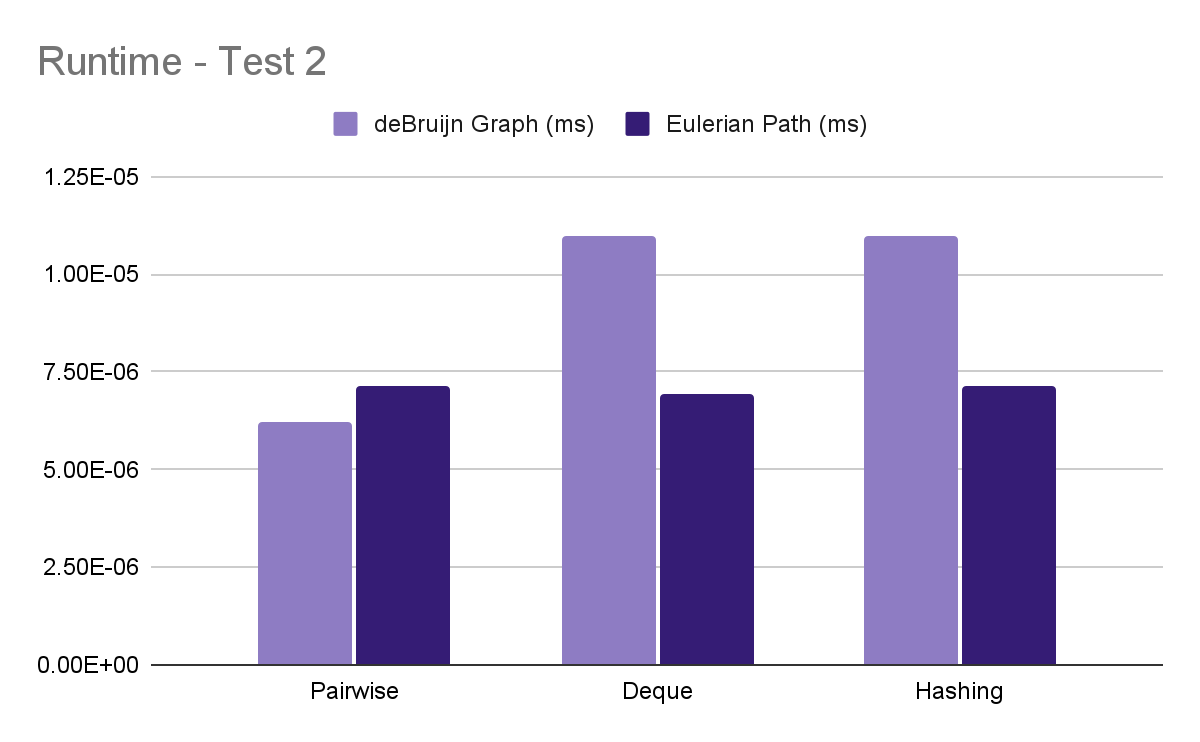
| DNA Sequence | deBruijn Graph (ms) | Eulerian Path (ms) | Assembled Sequence |
| --- | --- | --- | --- |
| Test 0: “aaaaaaaaaaa” | 2.9087066650390625e-05 | 7.867813110351562e-06 | Passed |
| Test 1: “agcagctcagc” | 1.1920928955078125e-05 | 6.198883056640625e-06 | Passed |
| Test 2: “agcagctcagg” | 1.0967254638671875e-05 | 6.9141387939453125e-06 | Passed |
| Test 3: random\_DNA\_sequence(10000, 20000) | 0.012624979019165039 | 0.005493879318237305 | Passed |

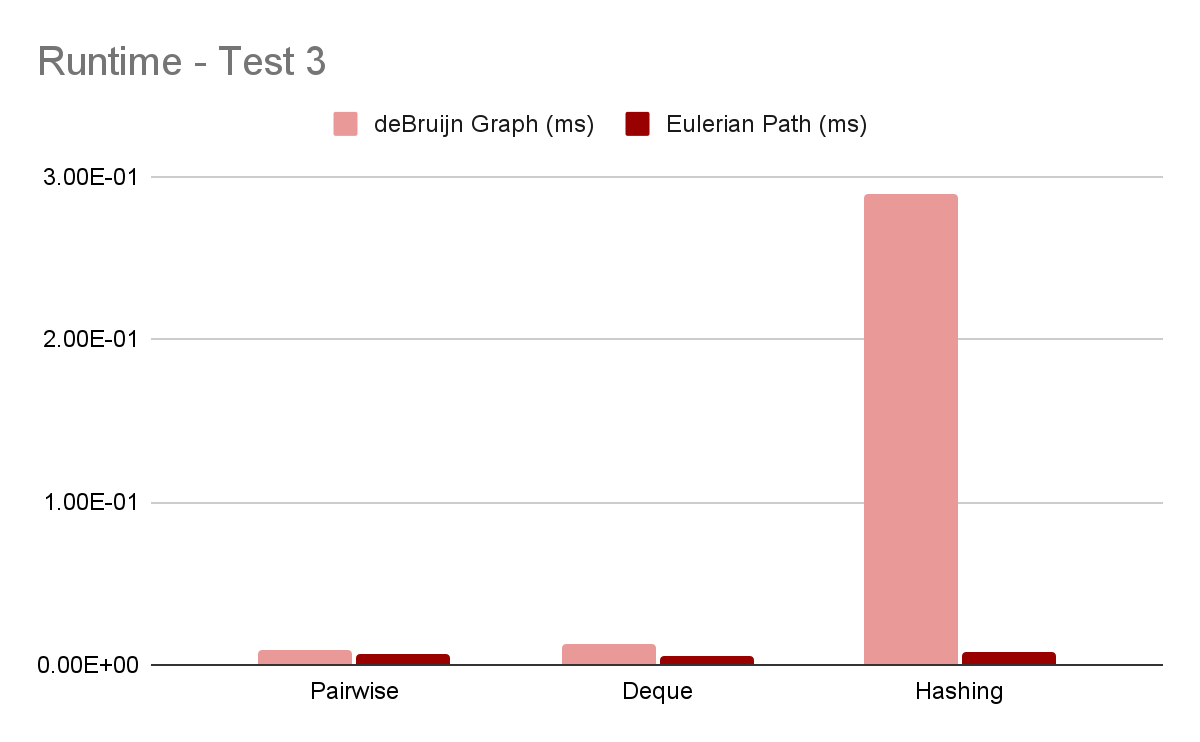
1. Testing k-assembler by k-mer hashing without deque

| DNA Sequence | deBruijn Graph (ms) | Eulerian Path (ms) | Assembled Sequence |
| --- | --- | --- | --- |
| Test 0: “aaaaaaaaaaa” | 1.2159347534179688e-05 | 4.76837158203125e-06 | Passed |
| Test 1: “agcagctcagc” | 1.1205673217773438e-05 | 5.245208740234375e-06 | Passed |
| Test 2: “agcagctcagg” | 1.0967254638671875e-05 | 7.152557373046875e-06 | Passed |
| Test 3: random\_DNA\_sequence(10000, 20000) | 0.28969883918762207 | 0.007423877716064453 | Passed |









**Discussion**

In comparing the performance of three methods for handling larger DNA sequences in reference to Test 3 results, it was found that the Pairwise method had the most efficient runtime of 0.0096 ms, while the hashing method with deque method had a runtime of 0.0126 ms, and the hashing method without deque had the slowest runtime of 0.2897 ms.

Upon analysis, it was observed that the use of deque in the hashing method improved its performance significantly compared to the non-deque version, cutting runtime by half, 0.2897ms to 0.0126ms. The deque data type in Python is introduced to solve the performance issues of using lists as stacks and queues, as appending and popping items from the right end of a list is usually efficient, but can become slower when the list needs to allocate more memory (Real Python, 2023).

The deque function was implemented to the debrujin\_graph\_from\_kmers() function and to the eulPath(path) function. In the debrujin\_graph\_from\_kmers(), to improve the performance of storing the list of outgoing edges for each node in a graph, we replaced a regular dictionary with a defaultdict object. This will create an empty deque object for each new key, which allows us to store the list of outgoing edges for each node in the graph using a deque. We then used the append() method to add each outgoing edge to the deque. In the eulPath(path), it adds a starting node to the deque object and repeatedly takes the leftmost node, gets its adjacent node, and appends it to the left side of the deque. It then removes nodes that have no more adjacent nodes and adds them to the path deque. The use of deque improves performance by efficiently removing and appending elements from both ends of the deque. This optimization results in improved performance compared to non-deque version.

However, when comparing all three methods, the pairwise method was ultimately found to be the most efficient in handling larger inputs. Therefore, it can be concluded that while the use of deque can improve the performance of the hashing method, the pairwise method is the preferred approach when dealing with larger DNA sequences. Graphs visualizing the runtimes can be found in supplemental files.

The time for the pairwise method was significantly fast compared to the time difference for the C++ skeleton. That meant that the pairwise function in python was not implemented correctly. The pairwise was so fast because it still used a dictionary. A new implementation of the pairwise function was made to run without any dictionary. The method needed a Node class that had the name of the kmer, the number of incoming nodes, and a list of outgoing nodes. The new pairwise function makes a node for every kmer and iterates over every kmer for each node to assemble the Node object. It outputs a list of Node objects. This new method gave more realistic times: The three short examples in test 1 took a fraction of a second, but the random sequence test of between 10000 and 20000 nucleotides can take as little as 28 seconds for a 10000 length sequence and up to around 2 minutes for a sequence of 20000 . This is a much slower method of building the deBruijn graph. Because of the slow run time for this pairwise function, there is no good reason to implement another Euler Path algorithm via this new graph data structure.

**Distribution of Work**

|  | Cynthia | David | Ryan |
| --- | --- | --- | --- |
| Task 1 | X | X |  |
| Task 2 | X | X |  |
| Task 3 | X | X |  |
| Conversion |  |  | X |
| Integration |  | X | X |
| Title |  | X |  |
| Abstract |  |  | X |
| Introduction |  | X |  |
| Methods | X | X | X |
| Results | X | X |  |
| Discussion | X | X | X |

**References**

1. Compeau, P., & Pevzner, P. (2018). *Bioinformatics algorithms: An active learning approach*. San Diego, CA: Active Learning.
2. Real Python (2023). *Python's deque: Implement efficient queues and stacks*, *Real Python*. Real Python. Available at: https://realpython.com/python-deque/
3. We would like to express special thanks to Professor Joe Song and doctoral student Yiyi Li for their invaluable assistance in helping us understand the algorithms and transitioning to Python. Their expertise and guidance have been instrumental in the successful completion of this project.