Int. J. Math. And Appl., 8(3)(2020), 125–139
ISSN: 2347-1557
Available Online: http://ijmaa.in/

International Journal of *Mathematics And its Applications*

Algorithmic Modeling of a Genome Sequence

Chiraag Kambalimath[1,*]

1 St. Stephen's Episcopal School, 6500 St. Stephen's Dr, Austin, United States.

Abstract: We describe how genomes are assembled and read. This involves the implementation of graph theory and the development of algorithms. The algorithms are written in Python.

Keywords: Genome Reconstruction, Algorithms, Bioinformatics.

1. Introduction

Deoxyribonucleic Acid, or DNA, is the basic foundation of all organisms. DNA resides in the nucleus of an organism's cell and expresses traits such as height, eye color, hair shape, disease risk, and much more. DNA is composed of two strands that combine to form a double-helix structure. Erwin Chargaff, Rosalind Franklin, James Watson, and Francis Crick were instrumental in discovering the double-helix shape of DNA. Each DNA strand consists of a series of nucleotides. Nucleotides are composed of three parts: a sugar (deoxyribose) group, a phosphate group, and a nitrogenous base. There are four types of nucleotide bases: adenine, thymine, cytosine, and guanine. Adenine always pairs with thymine, and cytosine always pairs with guanine. We illustrate DNA in the figure below. The brown lines represent the hydrogen bonds between nucleotides, the blue letters indicate the nucleotides of the "blue" strand, and the red letters illustrate the nucleotides of the "red" strand. Nucleotide sequences are very important because they code for an organism's traits. Given the illustration below, the nucleotide sequence of the red strand is ATCTAACGGTCAT. Both the number and sequence of nucleotide bases vary across different species and even organisms. We refer the reader to any biology textbook for further information about DNA and nucleotides. An example of such a textbook is [6].

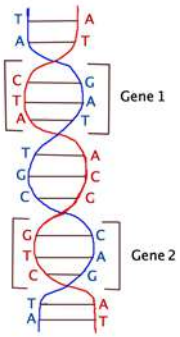


Figure 1. **Illustration of DNA and Genes.**

* *E-mail: chiraag.kambalimath@sstx.org*

DNA contains genes. Genes [7] represent portions of DNA and produce proteins that convey features like eye color and hair shape. In the figure below, we include brackets to illustrate genes. Each DNA section within the brackets represents a gene. In reality, genes contain many more nucleotide bases than portrayed below.

A genome is an organism's complete set of DNA. The human genome [3] consists of three billion nucleotides that form a sequence known as a genome sequence. We define a genome sequence and the process by which we obtain genome sequences in greater detail later in the paper. For now, we explore the field of bioinformatics and its role in genome reconstruction. Bioinformatics [1, 5] utilizes computer science in order to compile and analyze large data sets of biological information, including DNA. In this article, we explore the algorithmic component of bioinformatics as it relates to DNA sequencing [2]. In 2003, scientists determined the complete nucleotide sequence of the human genome as part of the Human Genome Project [4]. The Human Genome Project shed light on the importance of bioinformatics, as computer programs were essential in both obtaining the genome sequence and revealing its medical applications. This paper addresses an algorithm that is central to the assembly of genome sequences. The ability to analyze the human genome is critical in enhancing our understanding of genetic information [4]. For example, genome sequences can provide the location of certain genes and elucidate the genetic foundation of certain diseases. Moreover, examination of the genome reveals how genes collectively contribute to the makeup of an organism. With that said, genes only represent a fraction of the genome. As such, the ability to read an entire genome illuminates other significant DNA sections, including regulatory regions responsible for activating and deactivating genes as well as sections of DNA whose functions are currently unknown.

2. Genome Sequences

DNA exists in a three-dimensional shape; however, in this article, we think of DNA as two-dimensional. We do this in order to focus greater attention on the nucleotide sequence of each DNA strand. Consider Figure 2. On the left, we include a three-dimensional representation of DNA, and on the right, a two-dimensional depiction of DNA. For our purposes, both of these illustrations are the same in that they contain the same nucleotide sequences. As an example, for both illustrations in Figure 2, the strands depicted in red express the same nucleotide sequence: ATCTAACGGTCAT. Therefore, for the purpose of examining nucleotide sequences, the three-dimensional structure of DNA is irrelevant.

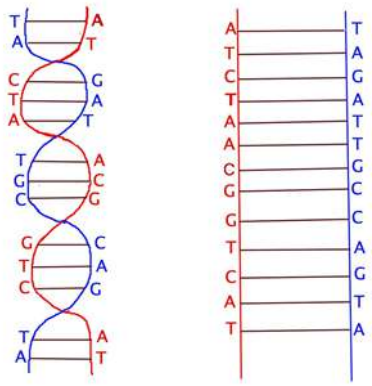


Figure 2. **Three-dimensional DNA and two-dimensional DNA**

As stated in the previous section, adenine always bonds with thymine, and cytosine always pairs with guanine. As a result, if we know the nucleotide sequence of one DNA strand, we also know the nucleotide sequence of its partner strand. As an example, consider either illustration in Figure 2. The red strand displays the following nucleotide sequence: ATCTAACG-

GTCAT. To determine the nucleotide sequence of the other strand, we simply replace each nucleotide in the red strand with its respective complement. The result is the following sequence: TAGATTGCCAGTA. Note that we do this without looking at Figure 2. However, if we look at Figure 2, we notice that the blue strand does in fact display the sequence TAGATTGCCAGTA.

Our overall goal is to read genomes. When we read a genome, or complete set of DNA, we try to figure out two things: the nucleotide sequence expressed by the genome and the order of that specific sequence. As discussed in the previous paragraph, we can focus our attention on only one strand of the genome because by knowing the contents of one strand, we automatically know the contents of the other strand. Furthermore, both the double-helix structure and the three-dimensional form of DNA is irrelevant to our main goal of reading genomes. As such, in this article, we think of a genome as simply a finite but long sequence of letters. Given that this sequence represents a genome, the only letters in the sequence are A,T,C, and G, which abbreviate adenine, thymine, cytosine, and guanine.

3. Sequencing a Genome

Almost every human cell contains an identical copy of a genome. Therefore, if one were to obtain a sample of skin or saliva, the sample would contain numerous identical copies of the human genome. Since the human genome contains three-billion nucleotides, it is very difficult to read and interpret as a whole. However, it is reasonable to read nucleotide sequences that are shorter in length. As such, in experiment, the human genome is divided into smaller sections through biochemical processes. These sections are then read and put back together so as to reconstruct the original genome. We illustrate this process in the explanation below. It should be noted that an algorithm must be used in order to reconstruct a specified genome. Consider our genome to be GCATAACTCTTAG. As mentioned above, after collecting a sample, we have multiple copies of the genome.

GCATAACTCTTAG

GCATAACTCTTAG

GCATAACTCTTAG

GCATAACTCTTAG

We then divide these genomes into smaller sections. In this particular example, we divide the genomes into sections that are four letters in length. However, the first and/or last sections of the genomes won't likely divide this way. The result appears as follows:

GCA TAAC TCTT AG

GC ATAA CTCT TAG

GCAT AACT CTTA G

G CATA ACTC TTAG

We disregard the sections that are not four letters in length, collecting only the sections that are four letters in length.

TAAC TCTT

ATAA CTCT

GCAT AACT CTTA

CATA ACTC TTAG

Each sequence above is a read. We use these reads to reconstruct the original genome. It should be noted that, in experiment, the sequences above are not provided to us in any particular order. Because of this, we write the sequences in lexicographic order below.

AACT ACTC ATAA CATA CTCT CTTA GCAT TAAC TCTT TTAG

From these reads, we hope to reconstruct the original genome.

4. Reconstructing a Genome by Overlapping Reads

Below, we are given the four-letter reads from the previous section.

AACT ACTC ATAA CATA CTCT CTTA GCAT TAAC TCTT TTAG

As previously stated, we want to use these reads to reconstruct a genome. We do this through overlapping. We explain this method below.

Consider the following read: AACT. As with regular words, the sequence AACT contains both a prefix and a suffix. Obtaining the prefix and suffix of a particular read is simple. To find the prefix of a read, we remove its last letter. Thus, the prefix of the read AACT is AAC. To find the suffix of a read, we remove its first letter. As a result, the suffix of the read AACT is ACT.

In order to reconstruct a genome through overlapping, we first obtain one of the four-letter reads from the list above. In this example, we choose the read AACT. In the figure below, we illustrate this in step one. We then obtain another read from the list above whose suffix is equivalent to the prefix of AACT. As such, we obtain the read TAAC. We place this read under AACT, making sure to align the letters that match. We show this in step two of the illustration. We continue this process by obtaining another read whose prefix is equivalent to the suffix of AACT. This is step three of the illustration. We now notice a pattern. Specifically, when we place a read below a given read, the suffix of the read below must be equivalent to the prefix of the read above. Oppositely, when we place a read above a given read, the prefix of the read above must be equivalent to the suffix of the read below. With this in mind, we form a stack that includes every four-letter read in the initial list. We see this in the final step of the illustration. Notice how each column in the final stack contains only one type of letter. Now that we have a stack, we can reconstruct the genome. To do this, we simply record the letter of each column. Once completed, we are left with the original genome. We see this in the final step of the illustration, where the reconstructed genome is boxed.

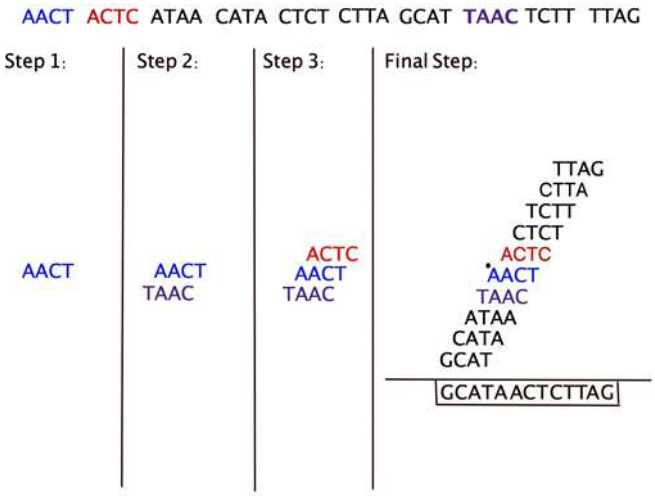


Figure 3. **Illustration of Overlapping Method**

While we were able to successfully overlap each read in this example, it should be noted that there are times when this can't be done. For example, given a list of reads, we may be unable to form a stack that includes every read in the list. Note also that when we increase the quantity of reads, the overlapping method becomes tedious in reconstructing a genome. Thus,

as mentioned in a previous section, we must use an algorithm to efficiently reconstruct a genome from a given list of reads. Describing this algorithm is the main purpose of this paper.

5. Defining the Computational Problem

As stated in the previous section, we want to reconstruct a genome from a given list of reads. In order to better understand how we do this, we introduce the following definitions and assumptions.

Definition 5.1. *A genome is a linear representation of the letters A,C,G, and T.*

For example, this is a genome: ATTAGCGCATGCTACCTA.

Definition 5.2. *A k-mer refers to a portion of a specified genome. The portion's length is defined by the numeric value of k.*

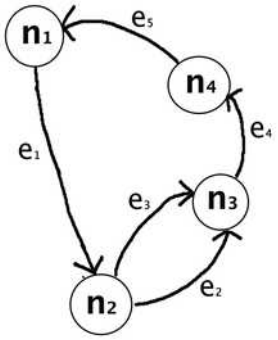
For example, TTAG is a k-mer of the genome ATTAGCGCA since it represents a portion of the genome. More specifically, we classify TTAG as a 4-mer given that it is four letters in length. It should be noted that TTAG is not the only 4-mer of ATTAGCGCA. To that end, a given genome contains multiple k-mers, so when we refer to a genome's k-mers, we are referring to a genome's complete list of k-mers. For example, the 4-mers of the genome ATTAGCGCA are ATTA, TTAG, TAGC, AGCG, GCGC, and CGCA. Each of these 4-mers make up the genome ATTAGCGCA. It should also be noted that a genome can contain repeat k-mers. Take the genome TAACGTAACGAT. In this genome, the 4-mer TAAC is found twice because it exists at two different sections of the genome.
Given the explanations above, consider the following situation. We put forth a list of k-mers and extract a genome that contains every k-mer in the list. In effect, we reconstruct a genome from its smaller sections. It should be noted that the variable k can represent any number; the list can also contain repeats and be in any order. As with all experiments, we can come across varying results. For example, while a list of k-mers can output more than one possible result, it can also return nothing at all. This article is solely concerned with results that contain at least one output. To make the task more concrete, we provide the following example: if we input the 3-mers CTG, TGA, ATC, TCT, GAC, and ACG, we want to output ATCTGACG because the 3-mers that make up the genome ATCTGACG are in fact ATC, TCT, CTG, TGA, GAC, and ACG (these 3-mers are equivalent to the 3-mers in the input, differing only in order). Note that we only gave one possible output. In the following example, we examine an input that returns more than one output. If we input the 3-mers TCC, CCG, CGC, GCC, CCA, CAC, ACC, and CCT, we should output both TCCGCCACCT and TCCACCGCCT. In this case, we constructed more than one genome.

6. Graph Theory

As previously stated, if we want to reconstruct a genome from a list of k-mers, we must use an algorithm. The algorithm that we use takes as input a list of k-mers and returns as output a genome that contains every k-mer in the input. Additionally, the algorithm is based on elements of Graph Theory. Graph Theory is a branch of mathematics that deals with the study of graphs. Below, we present elements of Graph Theory that are relevant to our goal of reconstructing a genome.

Definition 6.1. *A directed graph consists of vertices (nodes) and edges. Two nodes are connected by an edge, and the edges have directions. Given a pair of nodes (connected by an edge), one is classified as the head, and the other is classified as the tail. Edges point from their tail to their head.*

We depict nodes as circles and edges as arrows. Figure 4 displays a graph. We indicate the graph's nodes as n_1, n_2, n_3, and n_4, and we denote the graph's edges as e_1, e_2, e_3, e_4, and e_5. The table reveals the head and tail of each edge.



E	H	T
e1	n2	n1
e2	n3	n2
e3	n3	n2
e4	n4	n3
e5	n1	n4

Figure 4. **Illustration of a Graph**

Definition 6.2. *A path is a sequence of different edges* $(e_1, e_2, \ldots, e_n)$ *such that the head of* e_i *equals the tail of* e_{i+1}.

We illustrate a path in Figure 5. The path is e_1, e_2, e_3, e_4.

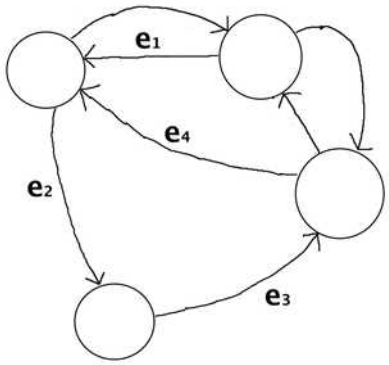


Figure 5. **Illustration of a Path**

7. Graph of a List of k-mers

For reasons that we explore later in the section, when given a list of k-mers, we construct a graph. The following observations and definitions help in understanding this.

Observation: Elements within a list may repeat themselves; however, in a set, each element is distinct (repeats may not occur). To avoid confusion, we depict lists between square brackets [], and we depict sets between curly brackets {}.

Definition 7.1. *The variable k is an integer greater than or equal to 2. The variable K represents a list of k-mers. A list may contain repeat k-mers. We define the set* $S(K)$ *to be the set of prefixes and suffixes of the k-mers in K.*

Consider the 4-mers in list K: [AACT, ACTC, CTCT, GCAT, TCTT, ACTC]. Notice that ACTC appears twice in the list. Because K is a list, this is acceptable. From each k-mer, we extract both its prefix and suffix. We illustrate this in the table above.

k-mers	prefix	suffix
AACT	AAC	ACT
ACTC	ACT	CTC
CTCT	CTC	TCT
GCAT	GCA	CAT
TCTT	TCT	CTT
ACTC	ACT	CTC

Figure 6. **Table of Prefixes and Suffixes**

The 3-mers depicted in red are the repeat 3-mers. Therefore, they will not be included in set $S(K)$. Set $S(K)$ will only contain the 3-mers depicted in black.

$$S(K) = \{AAC, ACT, CTC, TCT, GCA, CAT, CTT\}$$

Notation: Given a k-mer g, we denote its prefix and suffix as prefix(g) and suffix(g) respectively. For example, prefix(AACT) = AAC, and suffix(AACT) = ACT.

Definition 7.2. *k is an integer that is greater than or equal to 2. K is a list of k-mers. $S(K)$ is a set as defined in the previous definition. Given these properties, we define the graph $G(K)$ as follows.*

(1). For each k-mer in K, we create an edge whose label is equivalent to the chosen k-mer. Given an edge e, we denote its label as label(e).

(2). For each $(k-1)$-mer in $S(K)$, we create a node whose label is equivalent to the chosen $(k-1)$-mer. Given a node n, we denote its label as label(n).

(3). Given an edge e, its tail is the node t such that prefix(label(e)) = label(t).

(4). Given an edge e, its head is the node h such that suffix(label(e)) = label(h).

$G(K)$, as defined above, represents a graph of a list of k-mers.
To make this more concrete, we display the construction of a graph from list K.

$$K = [TTA, TAC, TAC, TAC, GTA, GGT, CTA, CGG, CCT, CCC, CAA, ATT, ACG, ACC, ACA]$$

From this list, we extract set $S(K)$.

$$S(K) = \{TT, TA, AC, GT, GG, CT, CG, CC, CA, AA, AT\}$$

The table in Figure 7 contains the information for the graph. Based on the information in the table, we draw a graph in Figure 8. We depict the labels of the nodes in black and the labels of the edges in red.
From the graph, we notice a loop. We also notice that more than one edge can connect two nodes. Furthermore, while the edges in the graph are labeled, they do not have to be; we can determine the label of an edge by examining the labels of its tail and head.

Label of Edge	Label of Tail	Label of Head
TTA	TT	TA
TAC	TA	AC
TAC	TA	AC
TAC	TA	AC
GTA	GT	TA
GGT	GG	GT
CTA	CT	TA
CGG	CG	GG
CCT	CC	CT
CCC	CC	CC
CAA	CA	AA
ATT	AT	TT
ACG	AC	CG
ACC	AC	CC
ACA	AC	CA

Figure 7. **Table for Graph**

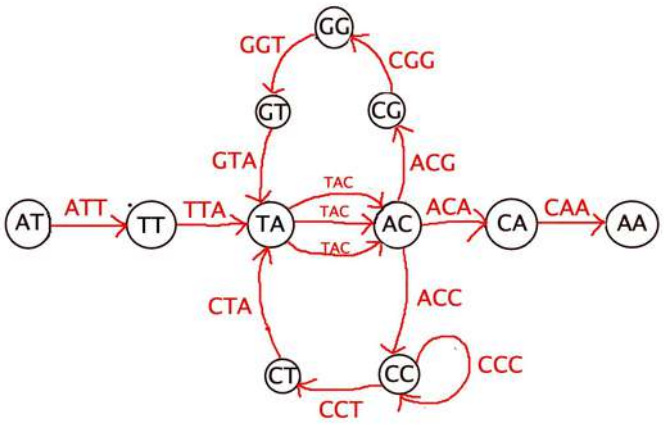


Figure 8. **Graph from k-mers**

8. The Genome of a Path from a Graph of K-mers

Definition 8.1. *Given a k-mer k, we denote the last letter of k as last(k). Given two sequences x and y, we denote the concatenation of x and y as $x + y$.*

For example, last(TCAGCT) = T, and TCAGCT + AC = TCAGCTAC.

Definition 8.2. *K is a list of k-mers. $G(K)$ is the graph of K. Given the graph $G(K)$, e_1, e_2,...,e_n is a path within the graph. We define the genome of this path as label(e_1) + last(e_2) + last(e_3) +...+ last(e_n).*

In Figure 9, we provide the same graph as in Figure 8 but label only the edges that make up the path e_1, e_2, e_3, e_4. These edges are labeled in red. Note that label(e_1) = TAC, last(e_2) = C, last(e_3) = C, and last(e_4) = T. Given this, we define the genome of path e_1, e_2, e_3, e_4 as TACCCT.

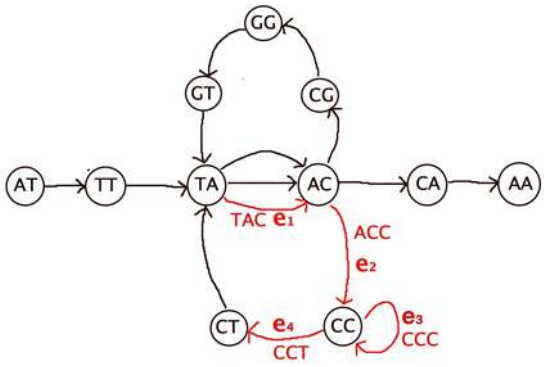


Figure 9. **Graph of a Path**

Observation: Let K be a list of k-mers. Let $G(K)$ be the graph of K. Within $G(K)$ the path e_1, e_2,...,e_n exists. The k-mers that make up the genome of this path are represented by the labels of each edge in the path.

We explore this observation in Figure 9. The genome of the path e_1, e_2, e_3, e_4 is TACCCT, and the list of 3-mers that make up this genome is [TAC, ACC, CCC, CCT]. When we examine and record the labels of each edge in the path, we get the same list of 3-mers. For example, the list [label(e_1), label(e_2), label(e_3), label(e_4)] = [TAC, ACC, CCC, CCT]. This observation is critical in determining the following results of this section.

Observation: Let K be a list of k-mers. Let $G(K)$ be the graph of K. Within $G(K)$ the path e_1, e_2,...,e_n exists. Assume that this path contains all the edges in $G(K)$. In this case, the k-mers that make up the genome of the path are equivalent to the k-mers in list K. The genome of this graph represents the solution to our main task of reconstructing a genome. As such, our genome reconstruction problem can be solved by finding a path within a graph that contains all of the edges of the graph.

To illustrate this, we provide a graph and its corresponding path in Figure 10. The graph is $G(K)$ and is constructed from the list K: [CTA,TTA, TAC, GTA, CGG, TAC, CCC, CAA, CCT, ATT, TAC, ACG, GGT, ACC, ACA]. In this example, the path e_1, e_2,...,e_{15} contains every edge in the graph. The genome of the path is ATTACCCTACGGTACAA. This genome contains the following list of k-mers: [ATT, TTA, TAC, ACC, CCC, CCT, CTA, TAC, ACG, CGG, GGT, GTA, TAC, ACA, CAA]. While in a different order, this list contains the same k-mers as list K, which we used to create graph $G(K)$.

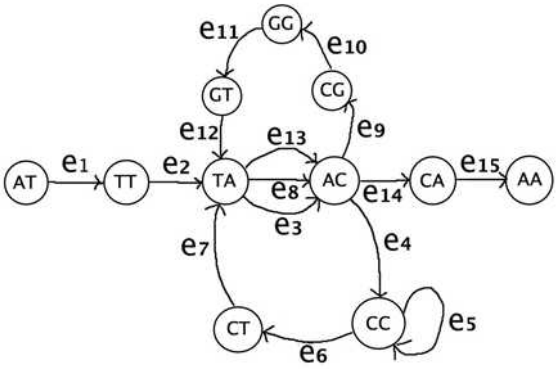


Figure 10. **Graph of a Complete Path**

It should be noted that more than one path can contain every edge in a given graph. Each distinct path expresses a different genome.

9. Eulerian Paths and Conditions for Their Existence

Let K be a list of k-mers. In the previous section, we reduced our task of genome reconstruction to finding a path within a graph that contains every edge in the graph. We now examine this further. A Eulerian path is a path that contains every edge in a graph. Therefore, if our goal is to reconstruct a genome from a given list of k-mers K, we must find a Eulerian path within a graph $G(K)$. It should be noted that while we can reconstruct more than one genome from a given list of k-mers, we are satisfied with the reconstruction of only one genome.

Definition 9.1. *Let n be a node in a graph. We define the indegree of n as the number of edges that have n as a head, and we define the outdegree of n as the number of edges that have n as a tail.*

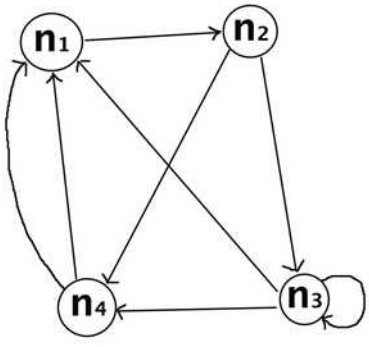
Given a node n, we denote the indegree of n as indeg(n) and its outdegree as outdeg(n). According to the graph in Figure 11, we list the indegree and outdegree of each node in the table. Notice in the table that the sum of indegrees is equivalent to the sum of outdegrees. This is always the case because each edge in a graph increases the indegree of its head by 1 while concurrently increasing the outdegree of its tail by 1.

Definition 9.2. *A graph is connected if for every pair of nodes (n_1 and n_2), there is a path that begins with n_1 as the tail of an edge and that ends with n_2 as the head of an edge.*

We provide an example of a graph that is connected and an example of a graph that is not connected in Figure 12.

Definition 9.3. *A path $(e_1, e_2, ..., e_n)$ is a cycle if the tail of e_1 is equal to the head of e_n.*

We provide a path that is a cycle and a path that is not a cycle in Figure 13. Both paths in each graph are illustrated in red.



Node	Indegree	Outdegree
n_1	3	1
n_2	1	2
n_3	2	3
n_4	2	2

Figure 11. **Indegree and Outdegree of a Given Graph**

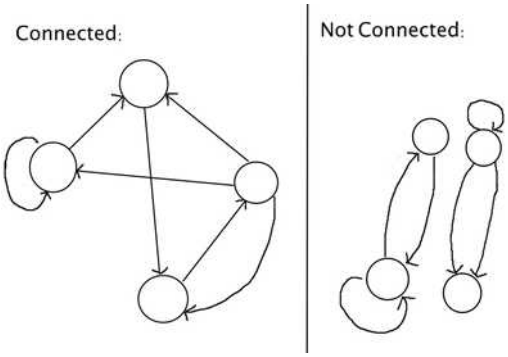


Figure 12. Illustration of a Connected Graph

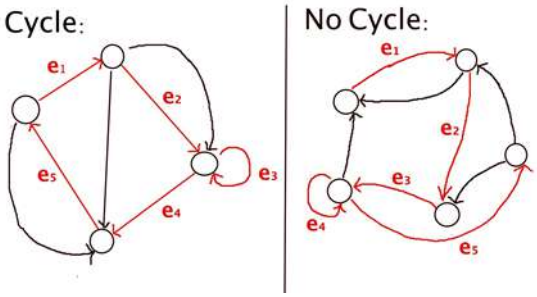
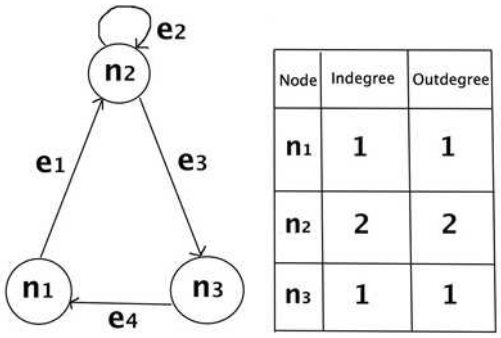


Figure 13. Illustration of a Cycle



Node	Indegree	Outdegree
n_1	1	1
n_2	2	2
n_3	1	1

Figure 14. Illustration of a Eulerian Cycle

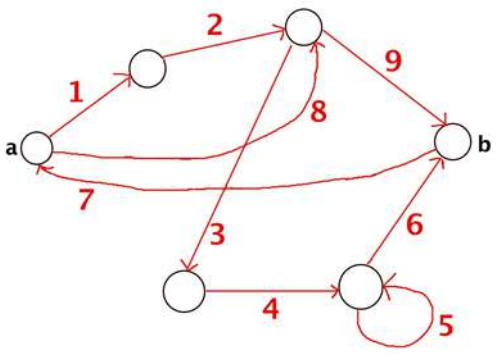


Figure 15. Illustration of a Eulerian Path

Theorem 9.4. *Let G be a directed graph with nodes V and edges E. Assume G is connected. We say that G contains a Eulerian cycle (a Eulerian path that is a cycle) if and only if for every node n, indeg(n) = outdeg(n).*

Figure 14 illustrates a graph where the path e_1, e_2, e_3, e_4 is a Eulerian cycle. Notice that the table shows indeg(n) = outdeg(n) for every node n.

Theorem 9.5. *Let G be a directed graph with nodes V and edges E. Assume G is connected. G has a Eulerian path that is not a cycle if and only if there exists a node a such that outdeg(a) - indeg(a) = 1 and a node b such that indeg(b) - outdeg(b) = 1. For every other node n, indeg(n) must equal outdeg(n). Additionally, every Eulerian path starts with an edge having a as its tail and ends with an edge having b as its head. Given these conditions, we illustrate a Eulerian path in Figure 15. We indicate the order of the edges in red.*

10. Algorithm to Find a Eulerian Cycle

In this section, we describe an algorithm that finds a Eulerian cycle within a graph. We assume that such a cycle exists, and as such, we assume that for any given node n, indeg(n) = outdeg(n). The graph in Figure 16 satisfies this condition. Now, we try to find a Eulerian cycle. First, we pick any edge and call it e_1. Then, we create a path within the graph and stop when we can no longer continue. We illustrate this in Figure 16.

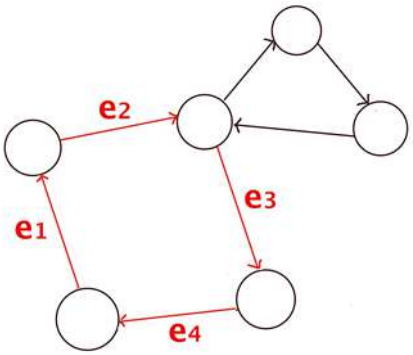


Figure 16. **Initial Graph**

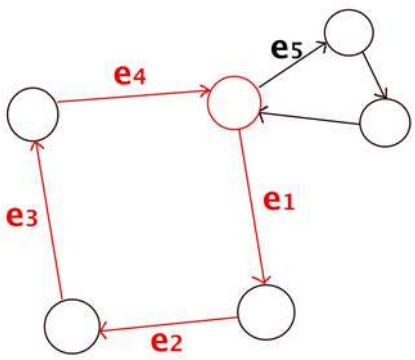


Figure 17. **Finding the Node**

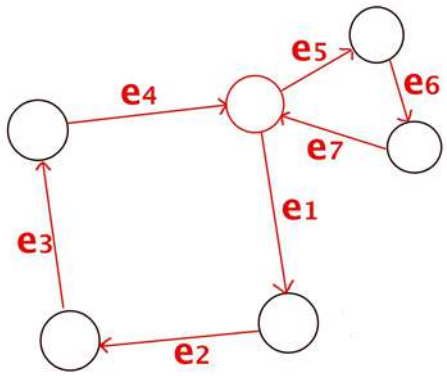


Figure 18. **Creating a Eulerian Cycle**

If our path contained every edge in the graph, we would be done. However, as illustrated below, this does not always occur. For example, the path e_1, e_2, e_3, e_4 does not include the edges depicted in black. Remember though that the graph is connected. As such, there should be at least one node in the graph that is the tail of a black edge. The next step is to find one of these nodes by going around the cycle and locating the first node that is the tail of a black edge. In Figure 17, we illustrate this node in red. Note that this node is in the cycle e_1, e_2, e_3, e_4 and is the tail of an edge not included in the path. We denote this edge as e_5 and re-trace the cycle. We begin at the red node and re-label the edges in order to maintain a path. We display these steps in Figure 17. Because we started at the red node when tracing this cycle, we must end at the red node. Furthermore, recall that we selected the red node because it is the tail of an edge (e_5) outside of the original path (e_1, e_2, e_3, e_4). Given this, we want to expand our cycle to contain both the red edges and the black edges. We illustrate this in Figure 18, which displays a Eulerian cycle. We are now done. If we did not end up with a Eulerian cycle, we would simply repeat the steps above until creating one.

11. Algorithm to Find a Eulerian Path

Our main goal is to find a Eulerian path, not a Eulerian cycle. In order to accomplish this, we first verify whether such a path exists within a given graph. The graph should contain a node a such that indeg(a) - outdeg(a) equals -1 and a node b such that indeg(b) - outdeg(b) equals 1. For every other node n, indeg(n) should equal outdeg(n). Next, we add an edge with node b as its tail and node a as its head. Now, for every node in the graph(n), indeg(n) = outdeg(n). This means we have a Eulerian cycle. We now find the Eulerian cycle within the new graph. This is illustrated in Figure 19.

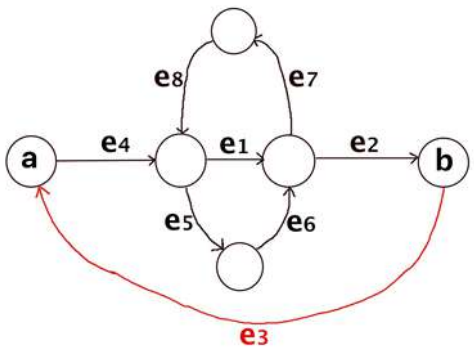


Figure 19. **Finding the Eulerian Cycle**

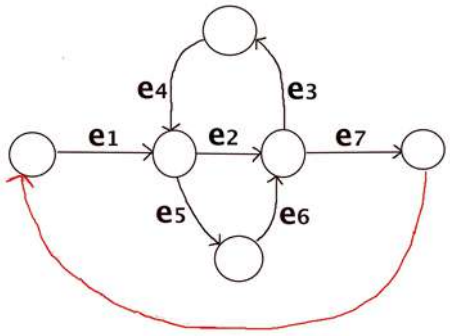


Figure 20. **Creating a Eulerian Path**

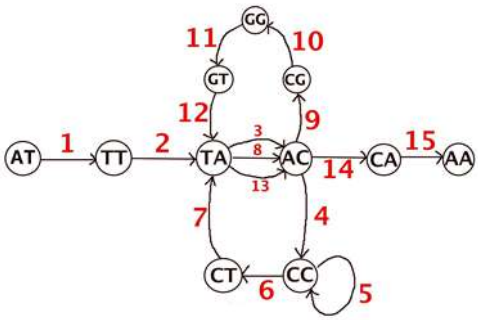


Figure 21. **The Genome ATTACCCTACGGTACAA**

Then, we retrace the cycle, beginning with the edge that comes after the added edge from node b to node a. We stop tracing once we reach node b. The result is a Eulerian path. We illustrate this in Figure 20 and 21.

12. Conclusion

The goal of this paper was to assemble a genome from a list of k-mers. Genome assembly is a legitimate computational problem because, as stated previously, the human genome can not be read and interpreted as a whole. We must split the genome into shorter sections, assembling the genome from these shorter sequences. In this paper, we referred to these smaller sections as k-mers. In order to actually reconstruct the genome, we developed an algorithm. Our algorithm drew from elements of graph theory. Through graphs, we were able to utilize Eulerian cycles and paths in order to achieve the task of genome reconstruction; from a list of k-mers, we assembled a genome. As referenced in the introduction, our task echoes the work of the Human Genome Project, which succeeded in the assembly of the entire human genome. As such, the work described in this paper applies to important biological and computational concepts and facilitates our understanding towards the complexities of DNA.

References

[1] Andreas D Baxevanis, Gary D Bader and David S Wishart, *Bioinformatics*, John Wiley & Sons, (2020).

[2] Terence A Brown, *DNA sequencing: the basics*, Number QP624 B88, (1994).

[3] Terence A Brown, *The human genome. In Genomes*, 2nd edition, Wiley-Liss, (2002).

[4] Leroy Hood and Lee Rowen, *The human genome project: big science transforms biology and medicine*, Genome medicine, 5(9)(2013).

[5] Arthur Lesk, *Introduction to bioinformatics*, Oxford university press, (2019).

[6] Kenneth Raymond Miller and Joseph S Levine, *Miller & Levine Biology*, Pearson, (2012).

[7] Charlotte K Omoto, Paul F Lurquin and Paul F Lurquin, *Genes and DNA: A Beginner's Guide to Genetics and its Applications*, Columbia University Press, (2004).