IJHSR
ijhighschoolresearch.org
RESEARCH ARTICLE

Reconstructing Genomes Using Graph-Based Algorithms

Aaron Kuang

Greenhill School, 4141 Spring Valley Road, Addison, TX, 75001, USA; aaron.kuang1228@gmail.com

ABSTRACT: Genome reconstruction is a crucial component of genome projects that aim to assemble an organism's complete genome sequence from short-fragmented DNA sequences. This task is fundamental and has wide-ranging applications in evolutionary research, medicine, conservation efforts, and synthetic biology. A common approach to genome reconstruction involves using graphs to identify overlapping sequences, known as k-mers, to recreate the entire genome. This study focuses on applying basic graph theory and Hierholzer's algorithm to efficiently identify Eulerian circuits and paths for genome assembly. The findings support the effectiveness of graph-based approaches as powerful tools in computational biology and genome research.

KEYWORDS: Graph Theory, Genome Analysis, Genetics, Genomics, Molecular Biology, Computational Biology.

Introduction

Genomic sequencing provides the foundation for understanding biological systems, from the genetic code of viruses to the human genome. A genome consists of the complete genetic information of an organism, encoded in DNA molecules. Each cell contains one DNA molecule, which is composed of long sequences of nucleotides forming a double helix structure. There are four types of nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T). The order of these nucleotides determines the sequence of the genome, which can be thought of as a long string of these four letters. Genomes are typically billions of nucleotides in length.[1]

Reading the genomes of organisms has many important applications. It has been used to study the evolution and relationships among species, identify genes and their functions, and advance medical research and treatments.[2,3] Genomic information has also improved agricultural practices, enabling the breeding of crops that are more resilient and yield.[4] New applications continue to emerge rapidly.

Each cell of an organism contains an identical copy of its genome. Thus, when a sample such as a drop of blood is taken, it contains billions of copies of the genome. Because genomes are so large, they cannot be read in one piece. Instead, sequencing technologies break the DNA into shorter fragments, typically a few hundred nucleotides long. These fragments are called k-mers, where k is the fragment length. Sequencing experiments read these k-mers, producing a large dataset of short DNA sequences. The challenge is to reconstruct the full genome from this collection of k-mers—a computational task known as the genome reconstruction problem. [5,6]

The rest of this article is organized as follows: In Section 2, we define the genome reconstruction problem precisely. Section 3 introduces directed graphs and the concept of Eulerian walks, which are essential for the reconstruction method. Section 4 explains how a directed graph can be constructed from lists of k-mers obtained in sequencing experiments. Section 5 shows why genome reconstruction reduces to finding an Eulerian walk in this graph and describes an algorithm to do so. Section 6 works through a simple example of genome reconstruction. Finally, Section 7 concludes with remarks and future directions. More information on the algorithms we explore in this article for the reconstruction of genomes, as well as different aspects of bioinformation, can be found in the references.[7-10]

The Genome Reconstruction Problem

As discussed in the introduction, every cell in a sample contains an identical copy of the genome. Thus, sequencing experiments are carried out, where the sections read are part of several identical copies of the genome/ This allows us to reconstruct the original sequence. We also recall that a k-mer is a section of the genome with k letters.

Consider the illustrative example in Table 1. We have four copies of the genome ACCTGTGTACC. Real genomes are much longer, but we use this short, simple example to illustrate the ideas. Each copy is broken into 4-mers. We start from the first letter on the first copy, from the second letter on the second copy, from the third letter on the third copy, and from the fourth letter on the fourth copy. This produces the 4-mers shown in black in the second column of Table 1. Shorter fragments, shown in red, are discarded.

Table 1: All the 4-mers of the genome ACCTGTGTACCT (in black).

Whole genome	4-mers in black
ACCTGTGTACCT	ACCT GTGT ACCT
ACCTGTGTACCT	A CCTG TGTA CCT
ACCTGTGTACCT	AC CTGT GTAC CT
ACCTGTGTACCT	ACC TGTG TACC T

From Table 1, we collect the 4-mers: ACCT, GTGT, ACCT, CCTG, TGTA, CTGT, GTAC, TGTG, TACC from the experiments. The goal is to develop an algorithm that takes as input this set of 4-mers and gives as output the genome

DOI: 10.36838/IJHSR87.1

ACCTGTGTACCT, which is not known. Note that some of the k-mers can be repeated, as ACCT appears twice in the list.

Of course, the situation described above is an idealized situation in which all the k-mers read have the same length. In real experiments, not all the k-mers read will have the same length. In this article, we will assume an ideal situation in which all k-mers read have the same length.

Thus, the goal is to use the list of k-mers read to infer the whole genome. We formalize this statement as follows:

The genome reconstruction problem: Let k be a positive integer. We are given an input list of k-mers. The goal is to find a genome whose k-mers are the list we were given, if such a genome exists. This genome is the output of the algorithm we will describe in this article.

It is instructive to look at the k-mers of a genome differently. Consider the example of the genome ACCTGTGTACCT. This genome has 12 letters. One of the 4-mers of this genome is the section of 4 letters that starts at the first letter, i.e., ACCT. A second 4-mer of this genome is the section of 4 letters that start at the second letter, i.e., CCTG. A third 4-mer of this genome is the section of 4 letters that starts at the third letter, CTGT. We can continue and obtain all the 4-mers. The list of the 4-mers obtained this way is: ACCT, CCTG, CTGT, TGTG, GTGT, TGTA, GTAC, TACC, ACCT. Note that this list is the same list that we obtained before, but in a different order.

When we are given the list of k-mers, it will not be sorted as described in the previous paragraph. In fact, our goal is to sort the k-mers as they appear in the genome, i.e., to have them as the list presented in the previous paragraph. Once sorted as in the last paragraph, reconstructing the genome is easy; simply start with the first k-mer, then add the last letter of the second k-mer, then the last letter of the third k-mer, and so on. This observation is at the heart of the algorithm we describe in this article.

The observation of the last paragraph is illustrated in Figure 1. Once sorted, the second k-mer is listed below the first k-mer and shifted one position to the right. Thus, the last k-1 letters of the first kmer are aligned on top of the first k-1 letters of the second k-mer. Note that these letters are the same. Similarly, the third k-mer is listed below the second k-mer and shifted one position to the right, etc. The result is shown in Figure 1. The letters in each column are the same. The genome can be reconstructed by dropping the letter of each column to the bottom line.

Figure 1:

In the rest of the article, we explain the mathematics and the algorithm needed to solve the genome reconstruction problem.

■ Directed Graphs

Graph Theory is a field of mathematics [11]. In this section, we introduce the concepts of Graph Theory that are needed by genome reconstruction algorithms.

Definition of directed graphs:

A directed graph is a pair of sets V and E. The elements of V are vertices, also known as nodes. In this paper, we will refer to them as nodes. The elements of E are edges. Each edge connects two nodes and has a specific direction, starting from one node, known as the tail, and ending at a node known as the head. We think of the edge as an arrow pointing from its tail to its head. The tail and the head of an edge are not necessarily different nodes. In visual representations, nodes are shown as circles, and edges as lines connecting the tail to the head of the edge with an arrow pointing toward the head. Figure 2 shows an example of a directed graph.

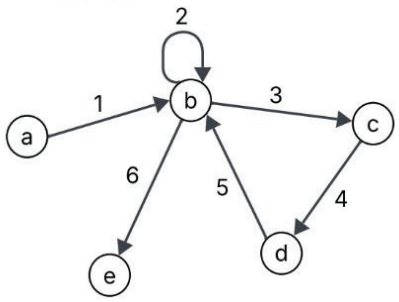


Figure 2: Example of a graph. The circles are the nodes. The arrows (not all straight) are the edges.

For reference, each node is labelled with a letter and each edge with a number. Table 2 lists the tail and head of each edge in Figure 2.

Table 2: List of the heads and tails of the edges of the graph of Figure 2.

Edge	Tail	Head
1	a	b
2	b	b
3	b	c
4	c	d
5	d	b
6	b	e

The graph of Figure 2 is a connected graph since it is in one piece. A disconnected graph, on the other hand, has two disjointed parts.

In-degree and out-degree of nodes:

The in-degree of a node is the number of edges that have that node as a head, and the out-degree of a node is the number of edges that have that node as a tail. In Table 3, we list the in-degree and out-degree of the Graph of Figure 2.

Table 3: List of the heads and tails of the edges of the graph of Figure 2. For each node of the graph of Figure 1, we list its in-degree, its out-degree, the edges that have the node as head, and the edges that have the node as tail.

Node	in-degree	out-degree	Edges with node as head	Edges with node as tail
a	0	1		1
b	3	3	1, 2, 5	2, 3, 6
c	1	1	3	4
d	1	1	4	5
e	1	0	6	

Walks and cycles:

A path is a sequence of edges where the head of one edge is the tail of the next. For example, the sequence of edges 1, 3, 4 is a path in the graph of Figure 2. Another path is e_1,e_2,e_2,e_3,e_4. These are the paths that particles can trace if they are only allowed to move along edges from the tails to the heads.

A walk is a path where each edge appears only once. For example, going back to the graph of Figure 2, the path 1, 3, 4 is also a walk, but the path 1, 2, 2, 3, 4 is not a walk because the edge 2 appears twice.

A cycle is a walk such that the tail of the first edge is the head of the last edge. This means that the particle moving along the cycle starts and ends at the same node. In Figure 2, the walk 3, 4, 5 is a cycle. The walk 2, 3, 4, 5 is also a cycle. The other cycle in that graph is the walk 3, 4, 5, 2.

Eulerian walks and cycles:

An Eulerian walk is a walk that contains all the edges in a graph. In the graph of Figure 2, the walk e_1,e_2,e_3,e_4,e_5,e_6 is an Eulerian walk. Note that not all graphs have Eulerian walks. We will later state a theorem providing the necessary and sufficient conditions for a graph to have an Eulerian path.

Naturally, an Eulerian cycle is a cycle that contains all the edges in a graph. Not all graphs have an Eulerian cycle. In fact, the graph of Figure 2 does not have an Eulerian cycle. Figure 3 shows an example of a graph with an Eulerian cycle. The cycle is e_1,e_2,e_3 is Eulerian.

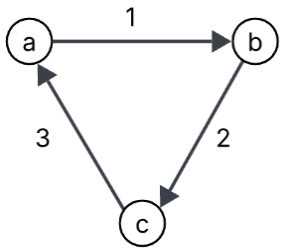


Figure 3: Example of a graph with an Eulerian cycle: 1, 2, 3.

The following theorem provides the necessary and sufficient conditions for a directed graph to have an Eulerian cycle.

Euler cycle theorem: For any directed, connected graph, there exists an Eulerian cycle if and only if, for each node in the graph, its in-degree and its out-degree are equal.

To illustrate this theorem, consider Table 4. The table shows that, for each node in the graph for Figure 3, its in-degree and its out-degree are equal. Thus, the graph has an Eulerian cycle.

Table 4: List of nodes of the graph of Figure 3 and their in-degree and out-degrees.

Nodes	In	Out	Difference
a	1	1	0
b	1	1	0
c	1	1	0

On the other hand, let's consider the graph of Figure 2. Note that the in-degree of node a is 0 and its out-degree is 1. They are different. Euler's theorem tells us that this graph has no Eulerian cycle. This is easily seen in a small graph, as the one in Figure 2, but when the graph has a large number of edges, Euler's theorem is useful in deciding if the graph has an Eulerian cycle.

For our purposes, we will need to find if a graph has an Eulerian walk that may or may not be a cycle. This leads to the next theorem, which is a natural extension of the Euler Cycle Theorem.

Euler walk theorem: A directed graph has an Eulerian walk that is not a cycle if all nodes have equal in-degrees and out-degrees except for two nodes. The out-degree of one of these nodes is equal to its in-degree plus one. This will be the starting node of the Eulerian walk. The out-degree of the other node of these nodes is equal to its in-degree minus one. This will be the ending node of the walk.

To illustrate the Euler walk theorem, consider the graph of Figure 2. The walk e_1,e_2,e_3,e_4,e_5,e_6 is an Eulerian walk. From Table 3, the in-degree of node a is 0, the out-degree of node a is 1; the in-degree of node e is 1, the out-degree of node e is 0; and for all other nodes, their in-degree equals their out-degree. The Euler walk theorem states that this graph has an Eulerian walk that starts at node a and ends at node e because the graph meets the conditions outlined above. In fact, e_1,e_2,e_3,e_4,e_5,e_6 is such an Eulerian walk.

Graph from a List k-mer

The input given is a list of k-mers, and the goal is to sort these k-mers into the sequence in which they appear in the genome we want to reconstruct, as in Figure 1. To that end, we will construct a graph and then find an Eulerian path on that graph. In this section, we explain how to construct the graph from the k-mers, and we also explain why finding an Eulerian path on this graph allows us to sort the k-mers and thus, reconstruct the genome.

We will explain and demonstrate the construction of a genome with an example. Suppose the list of k-mers that are given as input is ACCT, GTGT, GACC, CCTG, TGTG, ACCT, CTGT, GTGA, TGTG, TGAC. In this case, these are 4-mers. Our graph will have one edge for each of the 4-mers. Note that there may be different edges that have the same k-mer. This is the case in our example because the ACCT and TGTG appear twice in the list of k-mers.

Before describing the nodes, we remind the reader that the prefix of a k-mer is the (k-1)-mer that results from removing the last letter of the k-mer. For example, the prefix of ACCT is ACC. On the other hand, the suffix of a k-mer is the (k-1)-mer that results from removing the first letter of the k-mer. For example, the suffix of ACCT is CCT.

DOI: 10.36838/IJHSR87.1

The nodes will have (k-1)-mers associated with them. The (k-1)-mers of the tail node of an edge are the prefix of the k-mer of the edge, and the (k-1)-mer of the head node of an edge is the suffix of the k-mer of the edge. Unlike with the edges, there is no pair of nodes with the (k-1)-mer. For example, if an edge has the 4-mer ACCT, the 3-mer of its tail is ACC, and the 3-mer of its head is CCT. In Figure 4, we show just an edge, its head and its tail 4-mer, and the 3-mers of both the tail and head of the edge, but we do not show the rest of the graph.

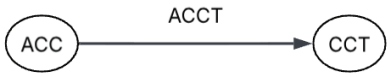


Figure 4: An edge and its 4mer. The head and tail of this edge and their 3-mers.

The rules described in the previous paragraphs determine the graph from the list of k-mers. For short, we will refer to this graph as the graph from the k-mers. The graph of our example, i.e., the graph of the list of k-mers ACCT, GTGT, GACC, CCTG, TGTG, ACCT, CTGT, GTGA, TGTG, TGAC, is shown in Figure 5.

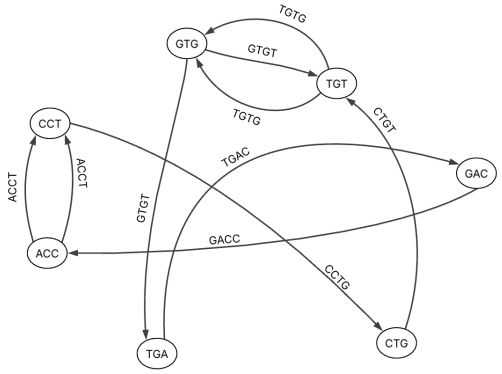


Figure 5: Graph of the list of 4-mers ACCT, GTGT, GACC, CCTG, TGTG, ACCT, CTGT, GTGA, TGTG, TGAC. The 4-mers of the edges and the 3-mers of the nodes are shown.

Assume we manage to find a Eulerian walk in this graph of Figure 5. Let e_1,e_2,..,e_n be this Eulerian walk. This means that the head of e_{i-1} is equal to the tail of e_i. The label of this node is both the suffix of the label of e_{i-1}, and also the prefix of the label of e_i. Thus, the last k-1 letters of the label e_{i-1} are the same as the first k-1 letters of the label of e_i. Thus, we have the k-mers sorted as previously explained, as in Figure 1, and the reconstruction of the genome from this point is immediate. To illustrate this claim, we show, in Figure 6, the Eulerian walk e_1,e_2,..,e_{10} in the graph of Figure 5. Figure 6 shows the same graph as in Figure 5. However, in Figure 6, we do not show the 4-mers of the edges and the 3-mers of the nodes. We show the labels of the edges. Assigning labels to the edges makes the description of the Eulerian path easier. Note that the labels can be chosen arbitrarily. To prevent the figure from becoming crowded, the 4-mers of each edge are shown in the table to the right of Figure 6. Note that we have not explained how to find Eulerian walks. This is done by applying the algorithm that we later explain in this article.

Edge	K-mer
e_1	TGTG
e_2	CTGT
e_3	TGAC
e_4	GACC
e_5	GTGA
e_6	ACCT
e_7	ACCT
e_8	CCTG
e_9	GTGT
e_{10}	TGTG

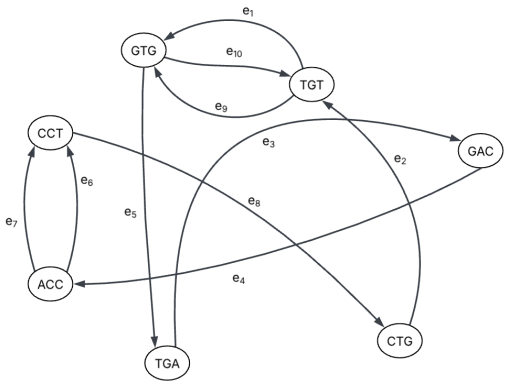


Figure 6: The left panel shows the same graph as in Figure 5.

We can list the 4-mers of the edges in the order of the path A, in this case ACCT, CCTG, CTGT, TGTG, GTGT, TGTG, GTGA, TGAC, GACC, ACCT. This is exactly the order in which the edges appear in Figure 1, from top to bottom. From where we can reconstruct the genome, as previously explained when we discussed Figure 1. This genome is ACCTGTGTGACCT.

Algorithm

As discussed in the previous section, our goal is to find an Eulerian path on the graph of k-mers. Before describing the algorithm to find such a path, we need to describe the algorithm to find an Eulerian cycle.

Algorithm to find an Eulerian Cycle:

In this section, we explain the algorithm we use to find an Eulerian cycle. We assume that, for all the nodes, their in-degree is equal to their out-degree. As previously described, this condition is necessary to guarantee that an Eulerian cycle exists.

There are two general algorithms to find an Eulerian cycle in a graph. The first is Fleury's algorithm, and the second is Hierholzer's algorithm, which is the method used in this paper. Hierholzer's algorithm is a stepwise construction by connecting cycles.

We explain the algorithm using the graph of Figure 7 as an example. We select an edge. We call this edge e_1. We create a walk with e_1 as its first edge. We extend the walk as much as possible. There may be more than one possible walk with e_1 as its first edge, but any walk will do. In our example, this walk we selected is the walk is $e_1e_2e_3e_4$, which is in green. Note that this walk can not be extended because there are no edges coming out of the last node, which is the head of the last edge, e_4 , that is not already part of the walk. Note that this walk is a cycle. This is a general fact, not just restricted to our example. This walk will always be a cycle.

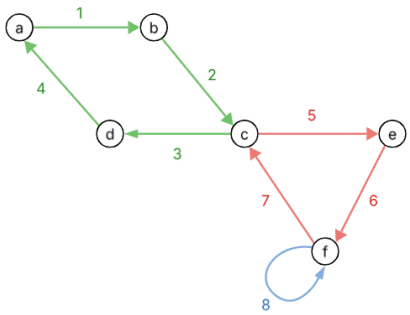


Figure 7: $e_1e_2e_3e_4$ is the first walk of Hierholzer's Algorithm shown in green.

If the cycle the algorithm obtained thus far by the algorithm had all the edges of the graph, this cycle would be Eulerian and would be the output of the algorithm. We would have accomplished our goal. In our example, the cycle $e_1e_2e_3e_4$, formed by the edges in green, does not contain all the edges of the graph and thus, it is not Eulerian.

If the cycle is not Eulerian, as in our example, the algorithm traces the cycle and stops at the first edge whose head is the tail of an edge not in the cycle. In our example, we start with the edge e_1 and travel to the node *b*. There are no edges coming out of *b* that are not in the cycle; this is the same as saying that there are no edges outside the cycle with *b* as a tail. We then travel through e_2 to the node *c*. The edge e_5 has *c* as a tail and is not in the walk. In other words, there is an edge not in the cycle that comes out of *c*. Thus, the edge e_2 is the edge in the cycle we were looking for.

The algorithm retraces the cycle, but starting at the edge following the one found in the previous paragraph. In our example, we start at the edge e_3. Thus, since we start at e_3, the cycle becomes is $e_3e_4e_1e_2$. Note that this walk can be extended because the endpoint of the cycle is now the node *c* and there are edges, e_5, not in the cycle yet, which have *c* as a tail. Thus, the algorithm can and does continue adding more edges to the walk.

The process continues as described in the last paragraph till all the edges in the graph are part of the cycle. Following our example, suppose the algorithm adds the red edges in Figure 7 to the cycle. Now the cycle is $e_3e_4e_1e_2\ e_5e_6e_7$. The algorithm is stuck again, but has 3 more edges than before. The whole process is repeated. The algorithm traces the cycle and stops at the edge e_6. It retraces the whole cycle starting from the next edge, the edge e_7, to get the cycle $e_7\ e_3e_4e_1e_2\ e_5e_6$. It can now continue and adds e_8. At this point, we have the Eulerian cycle $e_7\ e_3e_4e_1e_2\ e_5e_6e_8$.

Algorithm to find an Eulerian Walk:

Recall that our goal was to find an Eulerian walk, not necessarily an Eulerian cycle. In this section, we explain how the algorithm described in the previous section is extended to find Eulerian walks. There are two cases to consider.

First, if all nodes have equal in-degrees and out-degrees, the algorithm described in the previous section, Hierholzer's algorithm, finds an Eulerian cycle, which is also an Eulerian walk. No changes to the algorithm are necessary.

The second case is when all nodes have equal in-degrees and out-degrees except for two nodes. For one of the two nodes, its out-degree is equal to its in-degree plus one. We call this node the starting node. For the other node, its out-degree is equal to its in-degree minus one. We call this node the final node. The left panel of Figure 6 illustrates this case. *f* is the starting node and *a* is the final node.

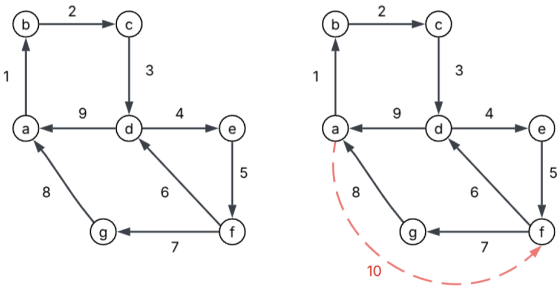


Figure 8:

We modify the graph by adding an edge that has the final node as the tail and the starting node as the head. We call this edge the added edge. In the example of Figure 8, this edge is *e*10, the edge in red in the right panel. Now, for all the nodes in this modified graph, their in-degrees equal their out-degrees.

We apply the algorithm explained in the last section to find an Eulerian cycle in the modified graph. Assume the cycle found in our example is $e_7e_8e_{10}e_4\ e_6e_9e_1e_2\ e_3e_4e_5$. We retrace the cycle till we find the added edge, e_{10} in our example. We now retrace the cycle starting at the edge that follows the added edge, i.e., starting at e_4 and ending at the edge before the added edge e_{10}, i.e., we stop at the edge e_8. This gives us the path $e_4\ e_6e_9e_1e_2\ e_3e_4e_5e_7e_8$. We have accomplished our goal. This is an Eulerian path of the original graph and is the output of the algorithm.

■ Example

In this section, we review the whole process of genome reconstruction by working out a small example.

We are given as input the list of *k*-mers ACCT, GTGT, GACC, CCTG, TGTG, ACCT, CTGT, GTGA, TGTG, TGAC. This is the same list of *k*-mers considered in Section 4. Note that *k*=4 in this example. Next, we construct the graph from this list of *k*-mers. This was also done in Section 4. The resulting graph is displayed in Figure 4.

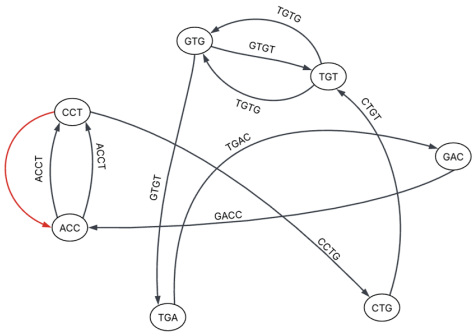
The next step is to find an Eulerian walk in the graph. We find this path using the algorithm described in Section 5. Instead of just displaying the result, we practice the ideas of the algorithm on this example by working out its steps in detail.

The algorithm first checks the in-degree and out-degree of each node. In our example, all the nodes have the same in-degree and out-degree, except for 2 nodes. The out-degree of the node with label ACC is equal to its in-degree plus 1. The in-degree of the node with label CCT is equal to its out-degree plus 1. This means that this graph has an Eulerian path.

Next, the algorithm creates a new edge from the node with out-degree smaller than its indegree, this is the node with label CCT to the node with in-degree smaller than its out-degree.

DOI: 10.36838/IJHSR87.1

We show the graph again in Figure 9, where we are also including this new edge in red. We call this new edge e_{11}.



Edge	K-mer
e_1	TGTG
e_2	CTGT
e_3	TGAC
e_4	GACC
e_5	GTGA
e_6	ACCT
e_7	ACCT
e_8	CCTG
e_9	GTGT
e_{10}	TGTG

Figure 9:

Next, the algorithm finds an Eulerian cycle in the augmented graph, i.e., the graph with the red edge. To accomplish that task, we first select any edge. We select e_3. We construct a walk starting at e_3. We are free to select any such walk. This walk is extended as much as possible. The walk we select is $e_3e_4e_6e_8e_2e_3e_9e_5$. This walk can not be extended because all the edges coming out of the head of the last edge of the walk, i.e., e_5, are already in the walk. An edge coming out of a node means that the node is the tail of the edge. The walk found is always a cycle.

Next, the algorithm travels through the walk and stops at the first edge whose head is the tail of an edge not in the walk if there is such an edge. The head of e_3 is the node whose label is GAC. The only edge coming out of this node is e_4, since this edge is already in the walk, we continue through the walk. The edge in the walk coming after e_3 is e_4. The head of e_4 is the node whose label is ACC. The edges coming out of this node are e_6 and e_7. The edge e_7 is not in the walk. We have found the first edge whose head is the tail of an edge not in the walk. This is the edge e_4. The algorithm re-traces the cycle, but starting with the edge coming after e_4, the edge found in this step. The cycle has become $e_6e_8e_2e_3e_9e_5e_3e_4$. Note that now, there is an edge not in the cycle coming out of the last edge in the cycle, e_4. Thus, the cycle can be extended. This is what the algorithm does. In our example, we obtain the cycle $e_6e_8e_2e_3e_9e_5e_3e_4e_7e_{11}$.

The process of the last paragraph is repeated. We find that the first edge of the cycle whose head has an edge that is coming out of it and is not in the cycle is the edge e_2. Thus, retracing the cycle but starting at the edge after e_2, the cycle becomes $e_3e_9e_5e_3e_4e_7e_{11}e_6e_8e_2$. Next, we extend the cycle to obtain $e_3e_9e_5e_3e_4e_7e_{11}e_6e_8e_2e_1e_{10}$.

Now that the cycle contains all the edges, the algorithm retraces the cycle till it finds the edge that was added to the graph of k-mers, the red edge, which in our example is the edge e_{11}. Finally, the algorithm retraces the cycle starting from the edge after e_{11}, and stops at the edge before e_{11}. In our example, this is the edge e_7. We end up with the path $e_6e_8e_2e_1e_{10}e_3e_9e_5e_3e_4e_7$. This is the Eulerian walk in the graph of k-mers.

Once the Eulerian path is found, the genome is reconstructed as follows. Start with the label of the edge in the Eulerian walk. In our example, this is the e_6. Its label is ACCT. Then, add at the end the last letter of the second edge. In our example, this edge e_8. Its label is CCTG, so its last letter is G. Thus, we add G at the end of ACCT to get ACCTG. We do the same with the third edge. This is the edge e_2. Its label is CTGT. Its last letter is T. We add this last letter at the end of ACCTG to get ACCTGT. This process is continued through the whole walk to get the genome ACCTGTGGCTAC… This is the genome we were looking for. This is the output of the algorithm.

Results and Discussion

Algorithm Performance:

The algorithm reconstructs the original sequence by traversing each edge exactly once, thereby preserving the overlap information encoded in the k-mers. In the test case, the reconstructed genome matches the true sequence, and execution completes in under 1 millisecond on standard hardware.

Ambiguities and Repeats:

In larger and more complex genomes, multiple Eulerian paths may exist, creating ambiguities in reconstruction. Such cases arise from repeated k-mers, palindromic sequences, or variability in read lengths. To resolve these issues in practice, genome assemblers employ additional strategies such as error correction, assigning edge weights based on read coverage, or incorporating paired-end read information. Notable examples include assemblers such as Velvet and SPAdes, which integrate these heuristics into their pipelines.

Scalability:

The present implementation is designed for clarity rather than raw performance. Its runtime scales linearly with the number of k-mers, but efficiency can be improved through the use of optimized data structures. For large-scale datasets, techniques such as de Bruijn graph compression and parallelization are essential to ensure computational feasibility.

Conclusion

We presented a clean, accessible Python implementation of genome reconstruction using Eulerian paths in de Bruijn graphs. This approach illustrates how fundamental concepts from graph theory can be applied to support biological discovery.

Our implementation assumes an idealized setting where reads are error-free, of uniform length, and cover the genome completely. In practice, sequencing introduces additional challenges: k-mers often vary in length, some k-mers may be missing while others appear multiple times, and read errors introduce noise. Addressing these issues requires incorporating statistical models, error correction, and abundance information.

Furthermore, while the algorithm described here demonstrates the essential idea, practical genome assembly demands more sophisticated data structures, graph compression techniques, and parallelization for scalability. These challenges continue to drive research in the fast-growing field of bioinformatics, where advances in algorithms and computational tools directly impact our ability to interpret complex genomic data.

References

1. Dale, J.W., Von Schantz, M. and Plant, N., 2011. From genes to genomes: concepts and applications of DNA technology.
2. Muffato, M., Louis, A., Nguyen, N.T.T., Lucas, J., Berthelot, C. and Roest Crollius, H., 2023. Reconstruction of hundreds of reference ancestral genomes across the eukaryotic kingdom. *Nature Ecology & Evolution*, *7*(3), pp.355-366.
3. Ouzounis, C.A., 2005. Ancestral state reconstructions for genomes. *Current opinion in genetics & development*, *15*(6), pp.595-600.
4. https://institute.global/insights/public-services/what-genomic-sequencing-and-why-does-it-matter-future-health
5. Gregory, T.R. ed., 2011. *The evolution of the genome*. Elsevier.
6. Liberles, D.A. ed., 2007. *Ancestral sequence reconstruction*. OUP Oxford.
7. Pevzner, P.A.; Tang, H.; Waterman, M.S. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA* **2001**, *98*(17), 9748–9753. https://www.pnas.org/doi/abs/10.1073/pnas.171285098
8. Compeau, P.E.; Pevzner, P.A.; Tesler, G. How to apply de Bruijn graphs to genome assembly. *Nat. Biotechnol.* **2011**, *29*(11), 987–991. https://www.nature.com/articles/nbt.2023
9. Kingsford, C.; Schatz, M.C.; Pop, M. Assembly complexity of prokaryotic genomes using de Bruijn graphs. *Genome Biol.* **2010**, *11*(12), R119. https://link.springer.com/article/10.1186/1471-2105-11-21
10. Pop, M., 2009. Genome assembly reborn: recent computational challenges. *Briefings in bioinformatics*, *10*(4), pp.354-366. https://academic.oup.com/bib/article/10/4/354/299108
11. Bollobás, B., 1998. *Modern graph theory* (Vol. 184). Springer Science & Business Media.

Authors

Aaron Kuang is a high school senior at Greenhill School. He is passionate about mathematics and computer science and enjoys developing algorithms for real-world problems.

DOI: 10.36838/IJHSR87.1