ESCI - UPF

GENOMIC ASSEMBLY

Algorithms and Data Structures

Greedy algorithms

Marta Ortigas - Xavier Crespo

Second year
19/03/2023

Contents

1 Introduction

You can find the submission files of this project at:
`https://github.com/XavierUPF/ADS-SUBMISSIONS-BDBI/tree/main/submission_2`

Overlap-layout-consensus (OLC) is a popular approach used in bioinformatics for genome assembly. In the absence of reference genomes, *de novo* genome assembly using OLC is a common approach for constructing a genome sequence. OLC works by identifying regions of overlap between reads and then using these overlaps to construct a directed graph called overlap graph. The goal of the OLC algorithm is to find a Hamiltonian path, a path in a graph that visits each vertex exactly once, in the overlap graph, which represents a contiguous sequence that incorporates all the input reads. The advantage of OLC is that it can generate long, contiguous sequences even when the individual reads are relatively short. However, the OLC algorithm is computationally intensive and can require a significant amount of memory to run, especially for large genomes.
For our toy-program, we will run tiny pieces of examples and explain what happens over the code.

2 *De novo* assembly and method explanation

De novo genome assembly is the process of reconstructing a genome sequence from scratch, without relying on a reference genome.

Imagine that we sequence a genome and as a result we have small chunks of DNA. But, remember, sequencing is done randomly, and we do not have any knowledge of the location from where these chunks belong. These chunks of DNA are what we call reads.

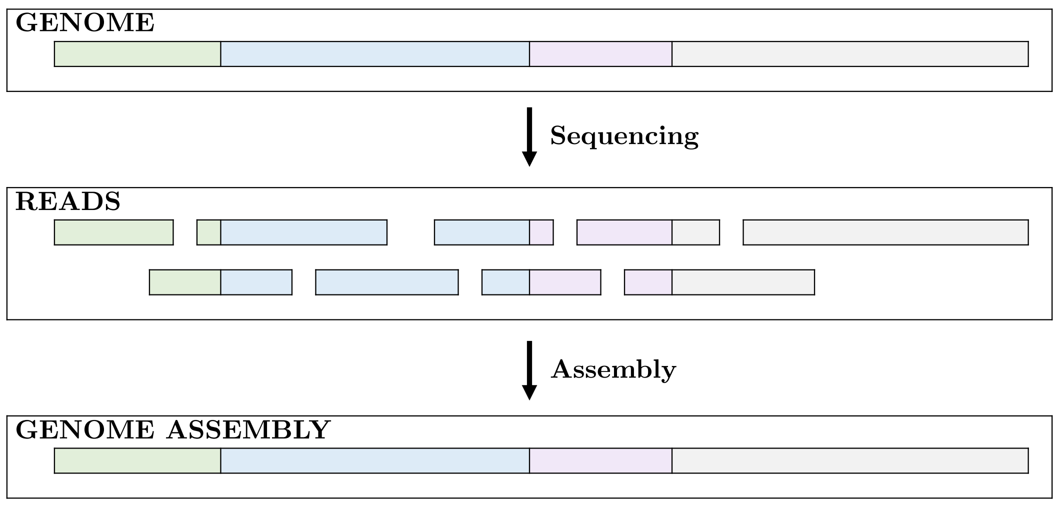

Figure 1: Diagram of own creation to depict *de novo* assembly. Each color represents a gene.

These reads are then compared to each other to identify overlapping regions. The overlaps are used to construct a graph called an overlap graph. In this graph, the nodes represent the reads, and the edges represent the overlaps between them.

The second step of OLC involves using the overlap graph to construct a layout graph. In this graph, the nodes represent the reads, and the edges represent the relative ordering and orientation of the reads in the final consensus sequence. The layout graph can be constructed using a variety of algorithms, including Greedy algorithms, dynamic programming, and graph theory algorithms.

The final step of OLC involves using the layout graph to generate a consensus sequence. This is done by traversing the layout graph to identify the most likely sequence that agrees with all the reads. The consensus sequence can be further improved by iteratively refining the layout graph and generating a new consensus sequence.

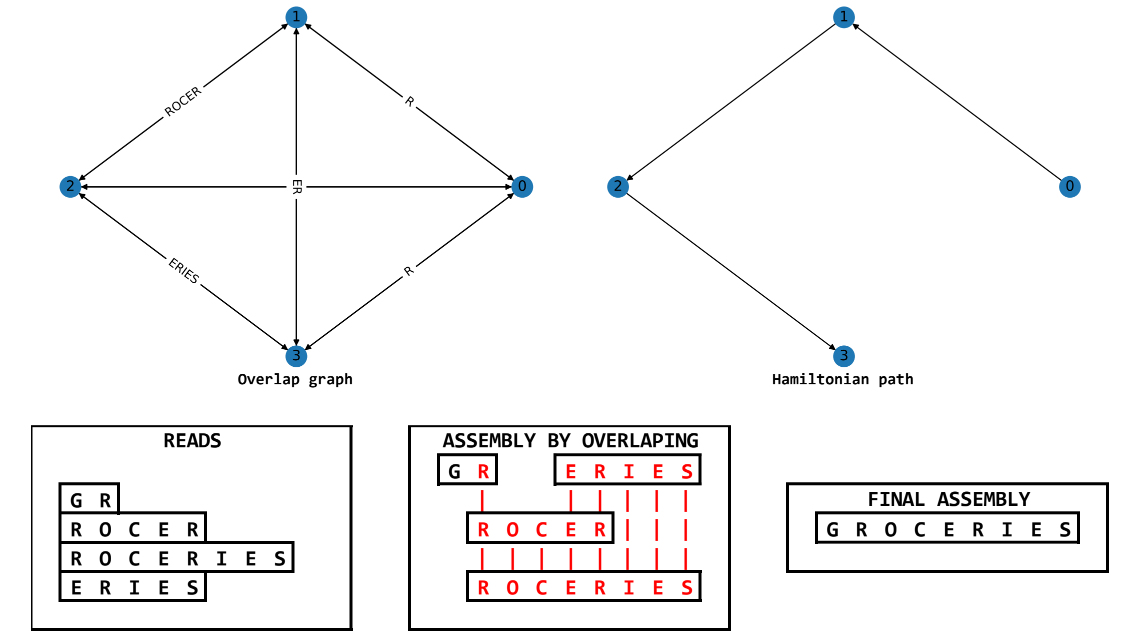

Figure 2: Toy example to show how the algorithm behaves.

3 Proof of concept

Here we have an example of an invented set of reads and expected output:

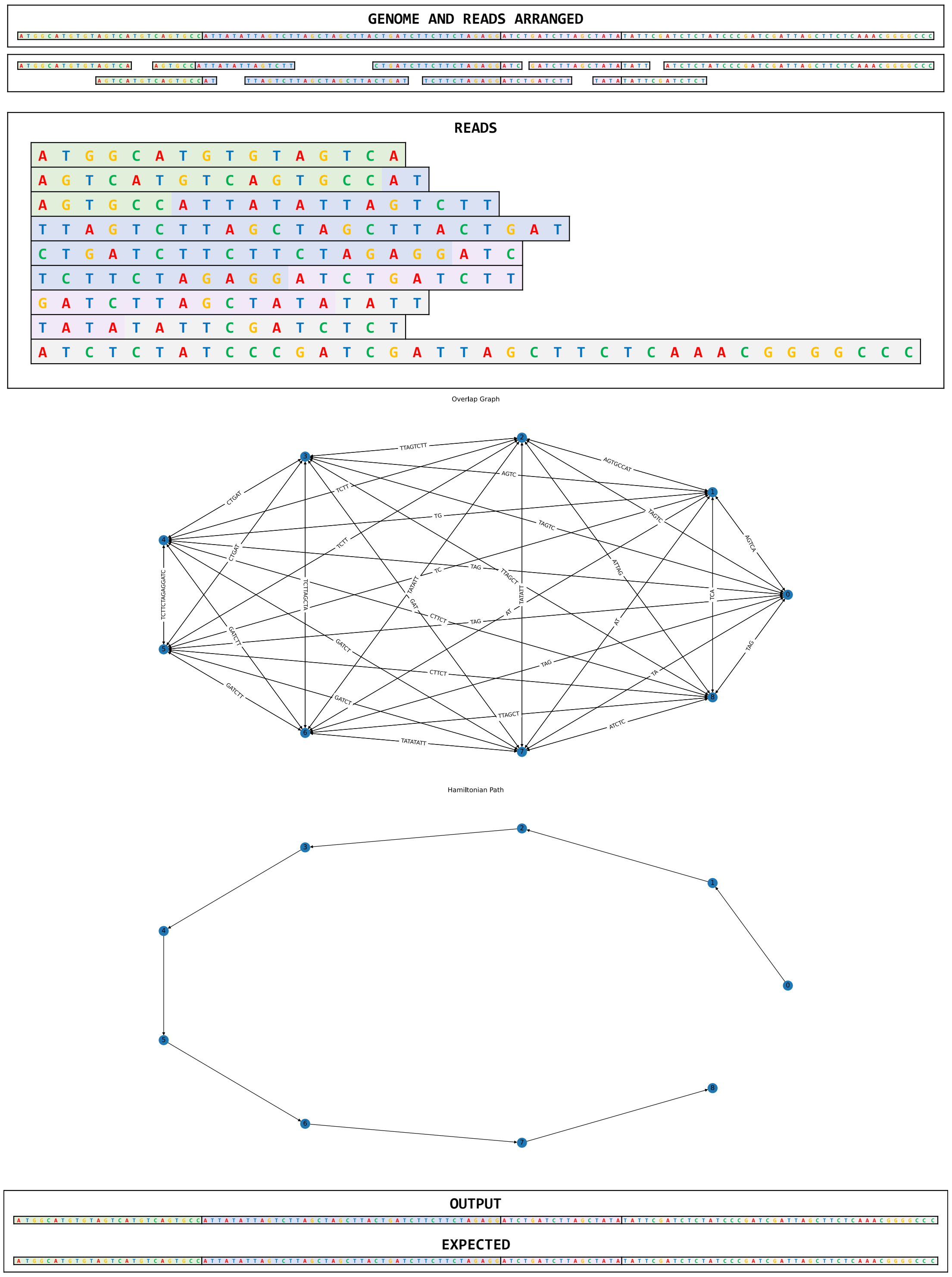

Figure 3

4 How the script works and conclusions

1. **Read Input Sequences:** The algorithm begins by taking input sequences in the form of DNA or RNA reads.
2. **Construct Overlap Graph:** The next step involves building an overlap graph using the input sequences. This graph is constructed by comparing every pair of sequences for overlap and creating an edge between them if there is a significant overlap.
3. **Find Hamiltonian Path:** The overlap graph constructed in the previous step is searched for a Hamiltonian path, which is a path that visits every node in the graph exactly once. In the case of OLC, this path represents the final reconstructed genome sequence.
4. **Generate Consensus Sequence:** Once the Hamiltonian path is found, the algorithm generates a consensus sequence by using a sliding window approach. A consensus base is calculated for each position in the window by choosing the most frequent nucleotide base observed at that position across all the reads.
5. **Output the Result:** Finally, the algorithm outputs the consensus sequence as the final result of the OLC assembly process.

As we have observed, when putting long reads and a big set of reads, it just takes ages, and it is not very precise mainly because it needs sufficiently enough reads. For this, other approaches such as Bruijn graph assembly (or overlapping K-mers as we know from class) may be a better solution. Still heavy in terms of computational power, but better.

It took us some time to figure out how we could manage the graphs. Turned out that networkx turned out to be quite handy for this task.