Life Sciences Outreach Fall Faculty Speaker Series
Teacher Professional Development Activity
Next-Generation Sequencing: Algorithms for Genome Assembly

Table of Contents

Activity Author 1
Overview 1
Objective 1
References 2
Activity 2

Activity Author

Nicholas Hilgert, PhD student in the Harvard Systems, Synthetic, and Quantitative Biology Program

Overview

This activity was designed for an audience of classroom high school biology teachers as part of a professional development program to further their knowledge in a research field. It has not been formally formatted or tested for a high school student audience because we believe that teachers are the best interpreters of content for their students. Therefore, we welcome teachers to adapt this activity for their own classroom needs.

Objective

During this activity, teachers will be exposed to algorithmic thinking in the context of bioinformatics. Using an artificial data set, they'll execute a simplified version of a popular algorithm used to assemble genomic sequences from next-generation sequencing data.

References

Adapted from: Compeau, Phillip EC, Pavel A. Pevzner, and Glenn Tesler. "Why are de Bruijn graphs useful for genome assembly?." *Nature biotechnology* 29.11 (2011): 987.

Some figures from Wikipedia.

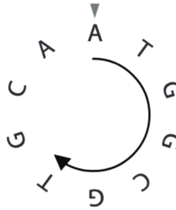
Introduction

The advent of novel sequencing technologies has generated data completely unlike any kind that scientists have had to deal with in the past. As a result, the tandem development of new algorithms for making sense of

this data has been essential to scientific progress. In this activity, we'll explore a popular way to represent and visualize DNA sequences in a computer, and we'll develop our own algorithm for putting them together.

Activity

Many next-generation sequencing technologies work by stitching several short snippets of a larger sequence together. In this activity, we'll suppose that we're sequencing the following plasmid, represented by the diagram below:



(Compeau et. al 2011)

The circular sequence in question (which we'll choose to start at the **`A`** marked by the arrow in the figure) is **`ATGGCGTGCA`**, but a next-generation sequencing workflow does not read this directly. Instead, it might output a few smaller subsequences from many copies of the plasmid. For example, the "data" could look something like the collection of sequences below, all of which are taken from the original one:

`CAATGGC` **`ATGGCGT`** **`CGTGCAA`** **`GGCGTGC`** **`TGCAATG`**

This is the raw information that the computer is responsible for making sense of. The question is, **given a set of short DNA sequences extracted from copies of a larger one, what is the larger DNA sequence?** As scientists, our job is to think of a program that will allow the computer to reliably answer this. And since a computer will need to execute whatever procedure we come up with, we'll need to specify it as a set of completely unambiguous instructions.

The algorithm

Such a set of instructions has a technical name: an algorithm. Though we typically give the instructions to a computer, humans can (and often do) execute algorithms, too; you don't need to be a computer to think computationally. For example, long division is an algorithm whose input is two numbers and whose output is a new number, their quotient. Elementary school students learn a set of bulletproof instructions that take them from input to output, even if they don't always execute them flawlessly. We want an algorithm here to take us from the set of short reads to the single long one. Together, we'll work through a popular algorithm used for this task.

Looking for patterns

Good algorithms are based on clever observations about the data on which they operate, and scientists are trained to make these observations by looking for patterns. One key observation that will underlie our algorithm is the following: the sequences have regions of overlap. Let's visualize this in the following table.

Problem 1: In the table below, fill each row with the sequence from the data that overlaps with the one above it. (For your convenience, the data is replicated below.)

`CAATGGC` ~~`ATGGCGT`~~ `CGTGCAA` `GGCGTGC` `TGCAATG`

A	T	G	G	C	G	T								
		G	G	C										

Notice that we've nearly solved the problem by hand! The original, longer sequence is just the sequence formed by listing the characters unique to each column. But keep in mind that this data is artificial; in real life we typically have many more than five sequences to arrange. Our brains found the pattern and successfully assembled the plasmid's code in this case, but this strategy isn't going to work more generally.

Luckily, there's still a lot to be learned here. Regions that *don't* overlap with one of the adjacent sequences (colored in blue and green) are only two characters long. The regions that *do* overlap with both adjacent sequences are three characters long. This is nothing but a more quantitative statement of our original observation! And quantifying this pattern makes it much more useful: subsequences of length two can contain information about the order in which the data should be arranged, since they indicate overlap with an adjacent data sequence. Our strategy for the algorithm will be to patch together two-letter subsequences with the three-letter ones that "link" them together.

Suppose this was the only knowledge we had of the structure of the data. Explicitly, suppose we only knew that two-letter sequences could help us arrange the subsequences, but not the specific ones. To cover every possibility, we could list every single candidate subsequence of length two. They're tabulated below:

AA	AT	TG	GG
GC	CG	GT	CA

Can you find all the three-letter subsequences? This just means we're looking for every unique three-letter sequence present in the data, to be linked by the two-letter ones.

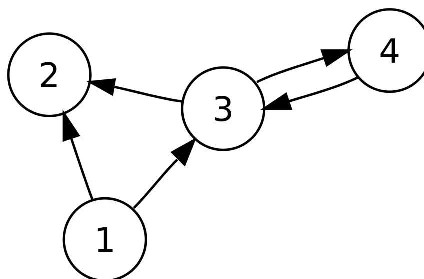
Problem 2: Find *every* distinct three-letter sequence contained in the data. List them in the table below—it should be completely filled. (No repeats, and order does not matter!)

`CAATGGC      ATGGCGT      CGTGCAA      GGCGTGC      TGCAATG`

ATG	TGG		TGC	
		CGT		GCA

Graphical thinking

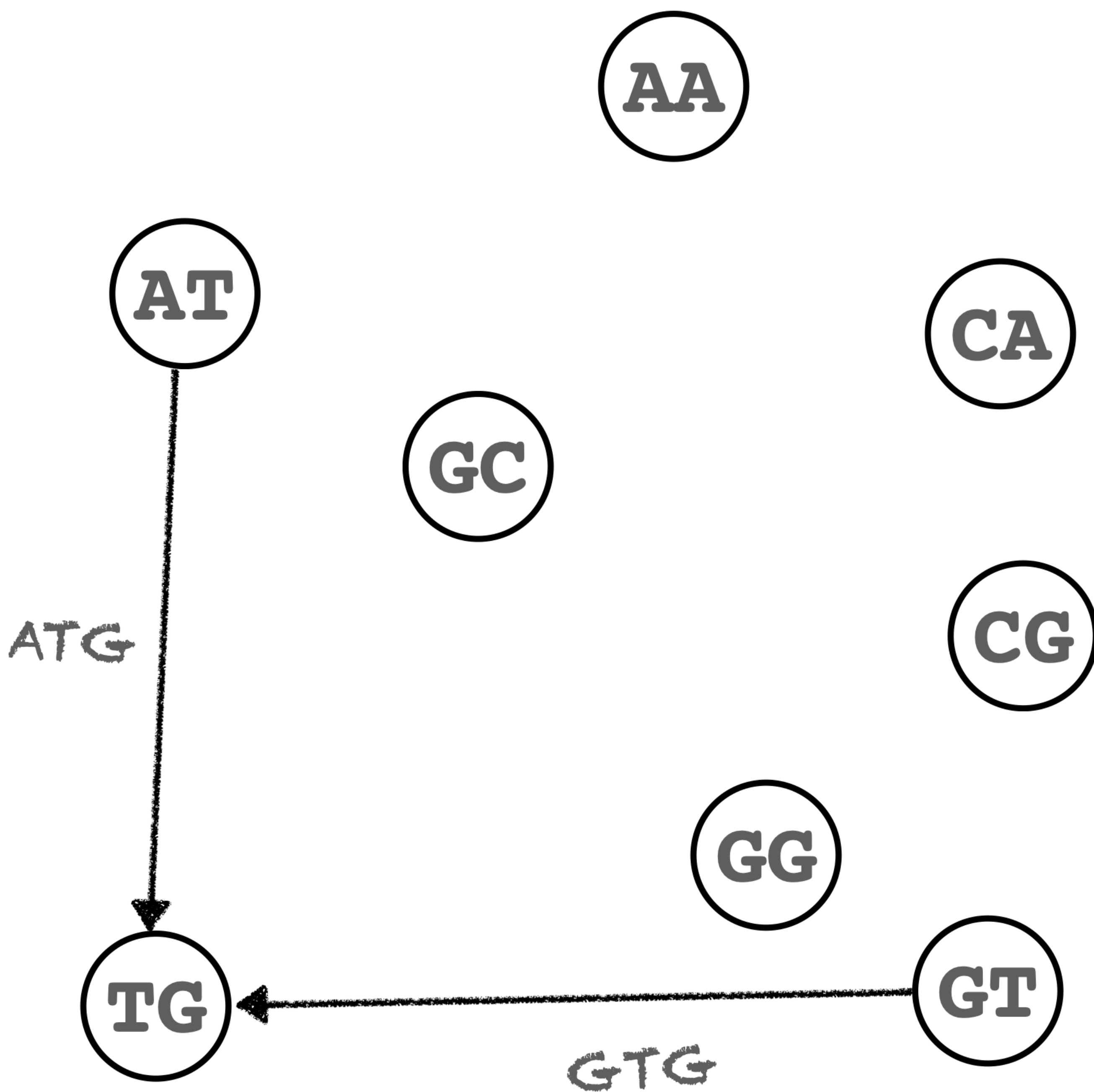
Now, we'll introduce a visual tool to help us finish out our problem. Many algorithms, including the one we're exploring, organize their logic with a graphical representation of the data. Here, a "graph" is just the technical word for a diagram like this:



(wikipedia)

A flexible way to represent nearly any kind of data, graphs are supported by a deep mathematical literature which we can use to prove facts about our data. They have two major components. First are the circles, or the nodes, which represent different data points, and second are the arrows joining them, or the edges, which represent a relationship between them. For example, Twitter could use a graph to represent the structure of a social network. Each account might correspond to a unique node, and an edge between two nodes might indicate that one account follows the other. In the example graph above, account 1 follows account 2, and accounts 3 and 4 follow each other.

We'll build a graph of our data to visualize our problem. In our case, **nodes will be the two-letter sequences** from above, and **edges will correspond to three-letter sequences**. I've included the nodes on the next page to get you started.

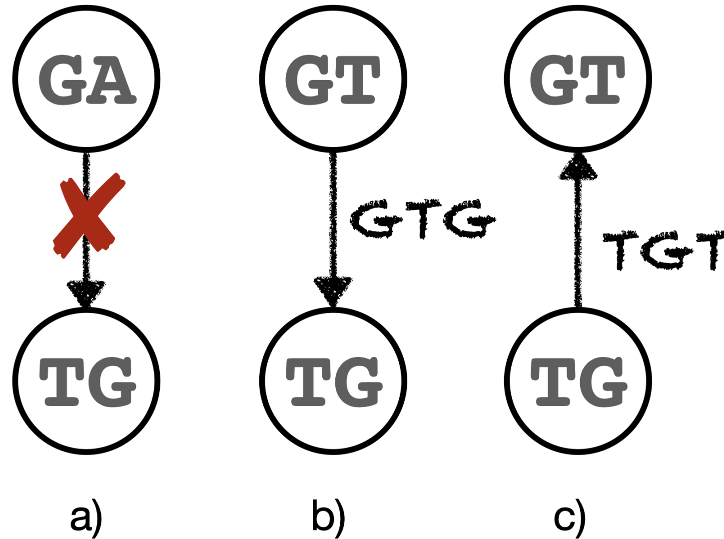


~~ATG~~	TGG	GGC	TGC	AAT
GCG	~~GTG~~	CGT	CAA	GCA

THE DEREK BOK CENTER
FOR TEACHING AND LEARNING
HARVARD

lso.

To complete the graph, we just need to fill in the edges, two of which are already drawn. But I haven't told you what relationship an edge should represent for this particular graph, so here are the rules for filling them in: **an edge should be drawn only when the two-letter sequences overlap by one character to form one of the data's three-letter sequences, in the order indicated by the arrow.**



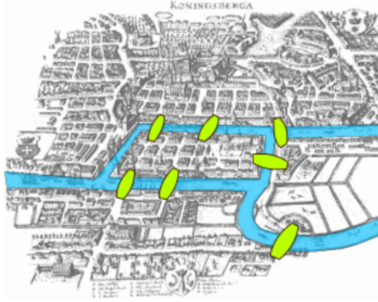
The above figure is included as a clarifying example. In **a)**, drawing an arrow from **GA** to **TG** is not valid since the second character in **GA** and the first character in **TG** are not the same (they don't overlap). If we fix this as in **b)**—where we've changed **GA** to **GT**—we get a valid edge. Notice that we could also try to flip the arrow as in **c)**, which, though seemingly legal, is not a valid edge for us since **TGT** is not one of our three-letter sequences from the data.

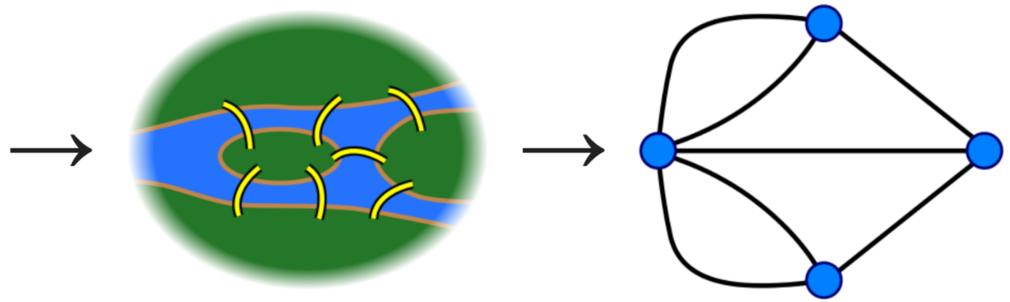
Problem 3: Fill in the rest of the graph, crossing out three-letter sequences from the bank as you fill them in.

Now that we've represented our data graphically, a mathematical fact *guarantees* that the original sequence can be directly reconstructed by traversing an eulerian path through the graph. We'll do so by looking at the overlap between the labels of the visited edges in the order we visit them.

Detour through the Bridges of Königsberg

In the mathematical study of graphs, a path that visits each edge only once is called an Eulerian cycle, named after the mathematician Leonard Euler. After looking at a map of a town named Königsberg (on the right), Euler was tasked with finding a tour of the city that crossed each bridge once and only once.





(wikipedia)

His insight was to formulate the question as a graph theory problem. To do so, he constructed a corresponding graph whose edges represent the bridges of Königsberg and whose nodes represent the pieces of land joined by the bridges, pictured on the right. Notice there are no arrows in this graph, meaning we are free to move in any direction across an edge or bridge. When he cast the problem in this language, he was able to prove that there is no cycle through the graph that visits each edge once, and so there is no tour through the city that visits each bridge once!

For graphs that *do* contain an Eulerian cycle (an easy condition to verify), there are efficient algorithms that bioinformaticians use to find them. Interestingly,the seemingly similar question of finding a path that visits each *node* once and only once belongs to the class of so-called NP-hard problems, a set of questions for which no known efficient algorithmic solution exists. A literal million-dollar question, the Clay Mathematical Institute will give a prize of $1,000,000 to anyone who can prove that efficient algorithms for NP-hard problems exist.

Reconstructing the sequence through an Eulerian cycle

Our problem is to find an Eulerian cycle through the sequence graph constructed above. More specifically, we'll try to find the one that reconstructs the original sequence. If we were computers, we'd need a step-by-step set of instructions to do so efficiently, and we'd use other data not included in this example to decide which cycle is the right one. Here is where the algorithm would take over and finish the job, but try to come up with your own solution!

Problem 4: find an Eulerian cycle through the sequence graph, and list the edges in the order you visit them in the table below. Note: you can start at *any* node you want! Remember, the DNA is circular, so the "start" and "end" are really up to you. Each column should correspond to a unique character—write it in the final row.

1												
2												
3												
4												
5												
6												
7												
8												
9												
10												