CAPSTONE

ALGORITHMS AND DATA STRUCTURES SPECIALIZATION

GENOME ASSEMBLY PROGRAMMING CHALLENGE

TABLE OF CONTENTS

The Story of 2011 European *E. coli* Outbreak

The start of the outbreak

What was the vehicle of the E. coli outbreak?

Crowdsourcing bioinformatics analysis of the pathogenic strain

Genome Sequencing

Exploding newspapers

The challenge of genome sequencing

Assembling phi X174 Genome

Phages

Assembling phi X174 genome from error-free reads

Assembling phi X174 genome from error-prone reads

Assembling phi X174 Genome Again

DNA arrays

Assembling phi X174 genome from k-mers

Assembling Puzzles from Repetitive Pieces

String Reconstruction as a Hamiltonian Path Problem

Overlap graph for k-mer composition

Universal strings

De Bruijn Graphs

From Bridges of Königsberg to Euler's theorem

Bridges of Königsberg

Eulerian cycles and genome sequencing

Euler's theorem

Constructing Eulerian cycles

Constructing Universal Strings

Splitting the Genome into Contigs

Genome Assembly Faces Real Sequencing Data

Inferring multiplicities of k-mers

Breaking reads into k-mers

Assembling error-prone reads

Assembling double-stranded DNA

How good is your assembly?

Assembling the Smallest Bacterial Genome

Assembling the *E. coli* X Genome

Assembling Genomes from Read-pairs

From reads to read-pairs

Transforming read-pairs into long virtual reads

From composition to paired composition

Assembling the *E. coli* X Genome from Read-pairs

FAQs

What Are Assembly Quality Metrics?

Data Formats for Representing Biological Sequences

Accessing Reads from the TY2482 Dataset

How Are Illumina Reads Generated?

How Are the Read-Pairs Generated?

PROBLEMS

1. What does it mean to assemble a genome?

2. Assembling phi X174 genome from error-free reads

3. What does it mean to assemble a genome from error-prone-reads?

4. Assembling phi X174 genome from error-prone reads

5. Assembling phi X174 genome from all *k*-mers

6. Puzzle assembly

7. Constructing de Bruijn graph from a set of *k*-mers

8. Linear-time search for an Eulerian cycle

9. Constructing a circular *k*-universal string

10. Contig Generation Problem

11. Circulation Problem

12. Selecting optimal *k*-mer size for constructing the de Bruijn graph

13. Bubble Detection Problem

14. Tip Removal Problem

15. Assembling phi X174 genome from error-prone reads using de Bruijn graphs

16. Assembling *N. deltocephalinicola* genome from simulated error-free reads

17. Assembling *N. deltocephalinicola* genome from simulated error-prone reads

18. Assembling *N. deltocephalinicola* genome from real reads

19. Assembling *E. coli* X genome from simulated error-free reads

20. Assembling *E. coli* X genome from simulated error-prone reads

21. Assembling *E. coli* X genome from real reads

22. String Reconstruction from Read-Pairs Problem

23. Assembling the *E. coli* X genome from error-free read-pairs with exact distances

24. Assembling the *E. coli* X genome from error-free read-pairs with inexact distances

25. Assembling the *E. coli* X genome from error-prone read-pairs with inexact distances

26. Assembling the *E. coli* X genome from real read-pairs

The Story of 2011 European *E. coli* Outbreak

The start of the outbreak

In April 2011, hundreds of people in Germany were hospitalized with **hemolytic uremic syndrome (HUS)**, a deadly disease that often starts as food poisoning with bloody diarrhea and can lead to kidney failure. German health officials immediately informed the World Health Organization about the incident, but they did not know that it was the beginning of the deadliest outbreak in recent history, caused by a mysterious bacterial strain that we will refer to as ***E. coli* X**. Within a few months, the outbreak had infected thousands and killed 53 people. To prevent the further spread of the outbreak, computational biologists all over the world had to answer the question "What is the genome sequence of *E. coli* X?" in order to figure out what new genes it acquired to become pathogenic.

What was the vehicle of the E. coli outbreak?

Researchers initially suspected contaminated food as the source of the outbreak, but they struggled to pin it down to a specific product and location. Shortly after the outbreak started, German authorities found traces of a suspicious bacterium in cucumbers imported from Spain. The vegetables were destroyed, and many Europeans stopped eating cucumbers. A month later, the European Commission backtracked, announcing that cucumbers had nothing to do with the outbreak (a German court would later rule that Spanish cucumber growers should be compensated for their financial losses). German health officials then linked the outbreak to a restaurant in Lübeck, where nearly 20% of the patrons had developed bloody diarrhea in a single week. By analyzing the meals eaten by the guests, researchers found that patrons who had eaten bean sprouts were much more likely to contract HUS.

Shortly afterwards, many people were hospitalized with HUS after eating sprouts in France, and researchers found that the same *E. coli* X strain was to blame. Scientists later tracked the source to a single lot of fenugreek sprout seeds imported from Egypt that had been sold two years before to distributors in Germany and France. After researchers found this link, Europe banned the import of fenugreek seeds from Egypt. Egyptian officials nevertheless argued that *E. coli* could not have survived for two years on seeds and that handling by the distributor could instead have resulted in sprout contamination. However, biologists know that *E. coli* can survive for years on seeds and still retain its pathogenicity.

Sprouts are usual suspects for *E. coli* outbreaks because they are cultivated in humid conditions, which support the growth of many bacteria. In fact, another *E. coli* outbreak afflicted nearly ten thousand children in Japan in 1996. However, the reaction to the 2011 outbreak was much swifter, thanks to genome sequencing methods and bioinformatics algorithms.

Crowdsourcing bioinformatics analysis of the pathogenic strain

On May 17, 2011, a 16-year-old girl, who had eaten a salad containing sprouts a week earlier, was admitted to an emergency room in Hamburg with bloody diarrhea. Doctors first suspected that the girl had been infected with a common pathogenic *E. coli* strain, which causes tens of thousands of hospitalizations worldwide annually and may lead to HUS. However, the blood sample from the girl did not pass the tests for known HUS-causing *E. coli* strains. At this point, biologists knew that they were facing a previously unknown pathogen and that traditional methods would not suffice – computational biologists would be needed to assemble and analyze the newly emerged pathogen.

To investigate the evolutionary origin and pathogenic potential of the outbreak strain, researchers started a crowdsourced research program (Rohde et al. 2011). They released bacterial DNA sequencing data from the girl in Hamburg, which elicited a burst of analyses carried out by computational biologists on four continents. They even used GitHub

https://github.com/ehec-outbreak-crowdsourced/BGI-data-analysis/wiki

for the project!

The 2011 German outbreak represented an early example of epidemiologists collaborating with computational biologists to stop an outbreak. In this Genome Assembly Programming Challenge, you will follow in the footsteps of the bioinformaticians investigating the outbreak by developing a program to assemble the genome of the deadly *E. coli* X strain. However, before you embark on building a program for assembling the *E. coli* X strain, we have to explain some genomic concepts and warm you up by having you solve a few simpler problems. Since genome assembly is such a difficult computational challenge, we provide a series of problems with gradually increasing complexity, starting with a small genome and simulated error-free reads. In the end, you will be well prepared to assemble a large *E. coli* X genome from real sequencing data. You can learn more about algorithms for genome assembly by reading the book by Compeau and Pevzner: *Bioinformatics Algorithms: An Active Learning Approach* (www.bioinformaticsalgorithms.org) or by attending the *Genome Sequencing* MOOC at Coursera, a part of Bioinformatics specialization (https://www.coursera.org/learn/genome-sequencing).

Genome Sequencing

Exploding Newspapers

Imagine that we stack a hundred identical copies of the New York Times newspaper on a pile of dynamite, and then we light the fuse. We ask you to further suspend your disbelief and assume that the newspapers are not all incinerated but instead explode into smoldering pieces of confetti. How could we use the tiny snippets of newspaper to figure out what the news was? We will call this conundrum the **Newspaper Problem** (Figure 1).

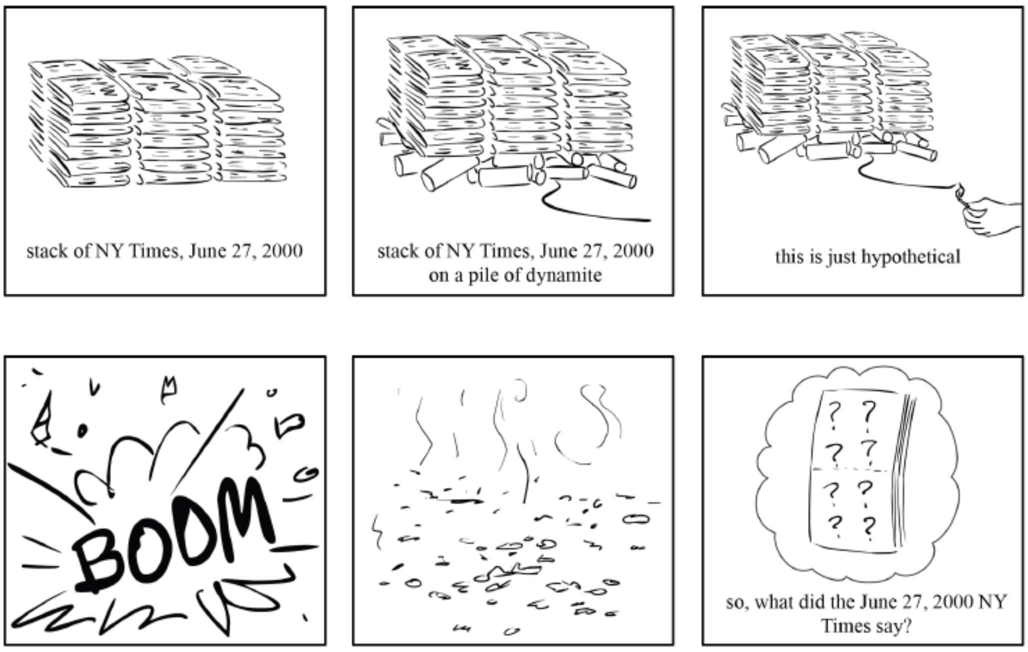
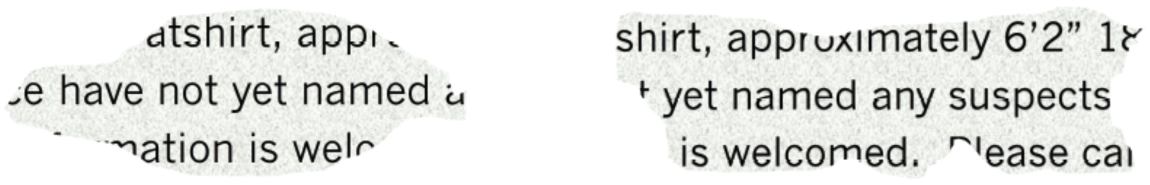

stack of NY Times, June 27, 2000
stack of NY Times, June 27, 2000 on a pile of dynamite
this is just hypothetical
BOOM
so, what did the June 27, 2000 NY Times say?

FIGURE 1. Don't try this at home! Crazy as it may seem, the Newspaper Problem serves as an analogy for the computational framework of genome assembly.

Because we had multiple copies of the same edition of the newspaper, and because we undoubtedly lost some information in the blast, we cannot simply glue together one of the newspaper copies in the same way that we would assemble a jigsaw puzzle. Instead, we need to use overlapping fragments from different copies of the newspaper to reconstruct the day's news, as shown in Figure 2.


atshirt, appr
e have not yet named a
mation is welc

shirt, approximately 6'2" 1
t yet named any suspects
is welcomed. Please ca

FIGURE 2. In the Newspaper Problem, we need to use overlapping shreds of paper to figure out the news.

Exploding newspapers is a good analogy for **genome sequencing**, determining the order of nucleotides in a genome. Genomes vary in length; your own genome is roughly 3 billion nucleotides long, whereas the genome of *Amoeba dubia*, an amorphous unicellular organism, is approximately 200 times longer! The first sequenced genome, belonging to a **phi X174** bacterial phage (a virus that preys on bacteria), has only 5,386 nucleotides and was completed in 1977 by Frederick Sanger, the inventor of DNA sequencing technology. Four decades after this Nobel Prize-winning discovery, genome sequencing has raced to the forefront of personalized medicine, as the cost of sequencing plummeted. As a result, we now have thousands of sequenced genomes, including those of many mammals.

The challenge of genome sequencing

To sequence a genome, we must clear some practical hurdles. The largest obstacle is the fact that biologists still lack the technology to read the nucleotides of a genome from beginning to end in the same way that you would read a book. The best they can do is to sequence much shorter DNA fragments called **reads**. Our aim is to turn an apparent handicap into a useful tool for assembling the genome back together.

The traditional method for sequencing genomes is illustrated in Figure 3. Researchers take a small tissue or blood sample containing millions of cells with identical DNA, use biochemical methods to break the DNA into fragments, and then sequence these fragments to produce reads. The difficulty is that researchers do not know where in the genome these reads came from, so they must use overlapping reads to reconstruct the genome. Thus, putting a genome back together from its reads, or **genome assembly**, is just like the Newspaper Problem.

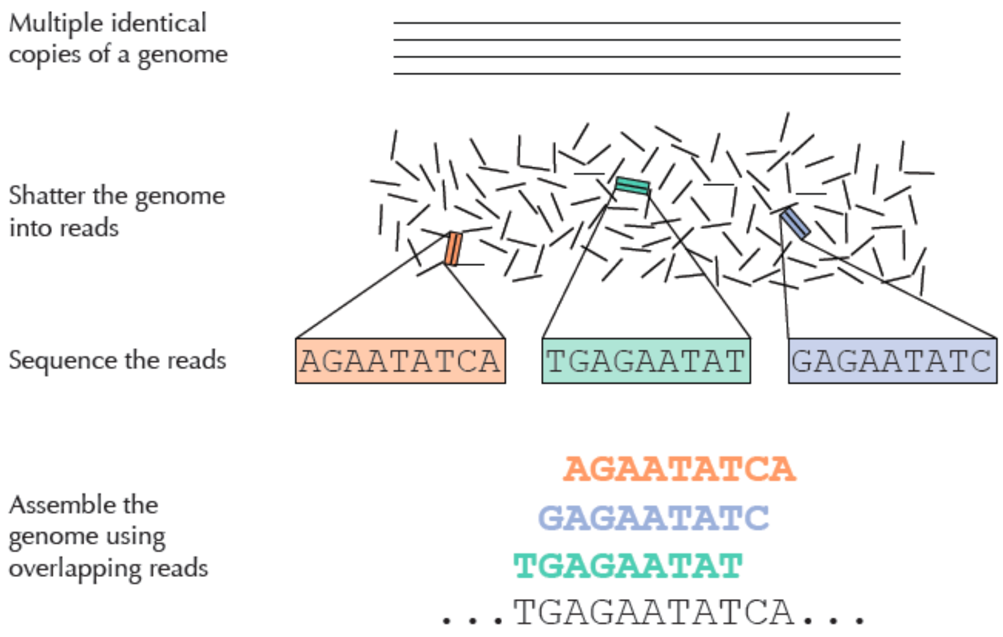

Multiple identical copies of a genome
Shatter the genome into reads
Sequence the reads
AGAATATCA
TGAGAATAT
GAGAATATC
Assemble the genome using overlapping reads
AGAATATCA
GAGAATATC
TGAGAATAT
...TGAGAATATCA...

FIGURE 3. Sequencing genomes by breaking them into short pieces (reads), identifying the sequence of nucleotides in each read, and assembling overlapping reads to sequence the genome.

Assembling phi X174 Genome

Phages

You will first follow in the footsteps of Fred Sanger to assemble the phi X174 phage genome. Phages are viruses that cannot replicate on their own and must infect bacteria to do so. Many phages are shaped like lunar landers (Figure 4), a design that helps them land on the cell wall of a bacterium and transmit their own DNA into the bacterial genome, so that when the bacterial DNA replicates, it creates new copies of the phage as well. Phages may provide benefits to the host bacterium by adding new functions to the bacterial genome. For example, harmless strains of bacterium *Vibrio cholerae* may be converted by phages into virulent ones, which cause cholera.

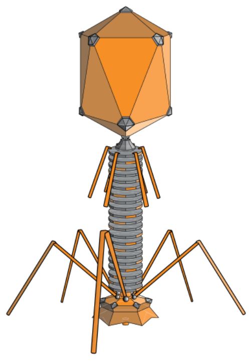


FIGURE 4. The structure of a phage.

Assembling phi X174 genome from error-free reads

We will provide you with 1000 simulated error-free reads randomly drawn from 5,386-nucleotide long phi X174 genome (each read is 100 nucleotides long). To add some suspense, all genomes in the simulated examples below will contain a single 10-nucleotide-long insertion (varying between different examples), and your goal will be to determine the sequences of these insertions, which we call **tags**.

At this point you are probably wondering what is the *exact* algorithmic problem we want you to solve. Indeed, previously in this specialization, you were facing well-defined algorithmic problem that the instructors have already formulated for you. But this is a capstone, and as in real life, you have to take initiative in your hands and start formulating the problems yourself!

What does it mean to assemble a genome? Formulate a rigorous algorithmic problem (provide Input and Output) that adequately models genome assembly.

STOP and Think: What problem from this course is the close "relative" of the genome assembly problem?

If you have difficulty answering this question, here is a hint. A read *R′* **overlaps** a read *R* if a sufficiently long suffix of *R* equals to a prefix of *R′*:

```
R       ATGCATGCACGTTGCTATGCCGATTCG
R'                GTTGCTATGCCGATTCGCCTGATTAGC
```

Given a set of reads, one can construct an **overlap graph** with vertices as reads and directed edges representing overlapping reads. We can define the length of each edge in this graph as the read length minus the length of the shared suffix and prefix. With this hint, we hope you can now answer the STOP and Think question above.

Assembling phi X174 genome from error-free reads. Assemble mutated phi X174 genome from error-free reads and find the inserted tag.

Many problems in this capstone will be similar to the problem above and will sound like "Assemble **some** genome from **some** reads." However, they will differ in complexity depending on the choice of genomes and reads. Since genome assembly is not for faint-hearted, we provide a series of problems with gradually increasing complexity so that, in the end, you will be well-prepared to assemble the **large** *E. coli* X genome from **real** reads. On the way towards this goal, you will have a chance to assemble **small** (phi X174), **medium-size** (smallest known bacterium), and **large** (*E. coli* X) genomes from simulated **error-free** or **error-prone** reads.

STOP and Think: Some reads have multiple overlaps, e.g. ATGATGATG and GATGATGAT have overlaps of lengths 7, 4, and 1:

```
ATGATGATG
  GATGATGAT
     GATGATGAT
        GATGATGAT
```

Which specific overlaps would you select for constructing the overlap graph?

Assembling phi X174 genome from error-prone reads

Next, you will assemble 1000 *error-prone* reads from a mutated phi X174 genome. For simplicity, we assume that all errors in simulated error-prone reads represent substitutions of nucleotides (i.e., no insertions or deletions).

What does it mean to assemble a genome from error-prone-reads? Formulate a rigorous algorithmic problem (provide Input and Output) that adequately models genome assembly from error-prone reads.

STOP and Think: How would you generalize the concept of the overlap graph for error-prone reads?

Assembling mutated phi X174 genome from error-prone reads. Assemble mutated phi X174 genome from error-prone reads and find the inserted tag.

STOP and Think: If you constructed the overlap graph and found a path in this graph in order to solve the problem above, then you have a chain of error-prone reads that follow each other in the (unknown) genome. How would you solve the problem of reconstructing the *accurate* genome sequence from a chain of error-prone reads?

Assembling phi X174 Genome Again

DNA arrays

Starting from Sanger's DNA sequencing approach that led to the phi X174 assembly in 1977, sequencing technologies went through a series of transformations that contributed to the emergence of personalized medicine a few years ago. By the late 1980s, biologists were routinely sequencing viral genomes containing hundreds of thousands of nucleotides, but the idea of sequencing a bacterial (let alone the human) genome containing millions (or even billions) of nucleotides remained preposterous, both experimentally and computationally. Indeed, generating a single read in the late 1980s cost more than a dollar, pricing mammalian genome sequencing in the billions.

In 1988, Radoje Drmanac, Andrey Mirzabekov, and Edwin Southern simultaneously and independently came up with an idea to reduce sequencing cost and proposed the futuristic and at the time completely implausible method of **DNA arrays**. None of these three biologists could have possibly imagined that the implications of his own experimental research would eventually bring him face-to-face with challenging algorithmic problems.

DNA arrays were invented with the goal of cheaply generating the set of *all k*-mers from the genome, in contrast to the original Sanger sequencing technology that generated *some* rather than all reads of length *k* from the genome. Given a string *Text*, its ***k*-mer composition** COMPOSITION$_k$(*Text*) is the **multiset** of all *k*-mer substrings of *Text*. For example,

COMPOSITION$_3$(TATGGGGTGC)={ATG, GGG, GGG, GGT, GTG, TAT, TGC, TGG}

Note that COMPOSITION$_k$(*Text*) lists repeated *k*-mers multiple times, e.g., GGG is listed twice in COMPOSITION$_3$(TATGGGGTGC). We have listed *k*-mers in the composition in the **lexicographic order** (i.e., how they would appear in a dictionary) rather than in the order of their appearance in TATGGGGTGC. We have done this because the correct ordering of *k*-mers along the genome is unknown when they are generated. We model genome assembly from *k*-mers as the String Reconstruction Problem:

String Reconstruction Problem: Reconstruct a string from its *k*-mer composition.
Input: An integer *k* and a multiset *Patterns* of *k*-mers.
Output: A string *Text* with *k*-mer composition equal to *Patterns* (if such a string exists)

Whereas Sanger's expensive sequencing technique generated 500 nucleotide-long reads, the DNA array inventors aimed at producing reads of length only 10, i.e., COMPOSITION$_{10}$(*Text*). At first, few believed that DNA arrays would work. In 1988, *Science* magazine wrote that, given the amount of work required to synthesize a DNA array, *"using DNA arrays for sequencing would simply be substituting one horrendous task for another."* It turned out that *Science* was only half right: in the mid-1990, Californian company Affymetrix perfected technologies for designing large DNA arrays, but DNA arrays ultimately failed to realize the dream that motivated their inventors because the value of *k* was too small to enable reconstruction of long genomes. Nonetheless, the failure of DNA arrays was a spectacular one: while the original goal (genome sequencing) dangled out of reach, an unexpected application of DNA arrays (analyzing genetic variations) emerged and transformed DNA arrays into a multi-billion dollar industry.

Assembling phi X174 genome from k-mers

You will now follow in the footsteps of the inventors of DNA arrays to assemble the phi X174 genome from its *k*-mers. We will provide you with all simulated error-free 10-mers from a mutated phi X174 genome and ask you to assemble it. Since you already know how to assemble 100-nucleotide long reads, assembling 10-nucleotide long reads should be easy, right?

STOP and Think. How would you define the concept of overlap between two short *k*-mers with respect to defining how long the overlapping prefix and suffix should be? For example, would you view 10-nucleotide long reads that share a 5-nucleotide long prefix and suffix as overlapping?

Assembling phi X174 genome from all *k*-mers. Assemble a mutated phi X174 genome from its 10-mers composition and find out the inserted tag.

You may find it surprising that this seemingly simple problem does not have a unique solution! The genome of phi X174 has two copies of 10-mer R_1=TGACGCAGAA (at positions 17 and 796) and two copies of 10-mer R_2=TTGATAAAGC (at positions 68 and 3831). If we represent phi X174 genome as a circular string $AR_1BR_2CR_1DR_2$ (A, B, and C, and D stand for genomic segments flanked by repeats R_1 and R_2), then $AR_1BR_2CR_1DR_2$ is not the only solution of the above problem. Indeed, $AR_1DR_2CR_1BR_2$ and $CR_1DR_2AR_1BR_2$ have exactly the same 10-mers as $AR_1BR_2CR_1DR_2$:

$$\begin{gathered} AR_1BR_2CR_1DR_2 \\ AR_1DR_2CR_1BR_2 \\ CR_1DR_2AR_1BR_2 \end{gathered}$$

The multiple reconstructions $AR_1BR_2CR_1DR_2$, $AR_1DR_2CR_1BR_2$, and $CR_1DR_2AR_1BR_2$ illustrate that repeated *k*-mers make genome assembly challenging. Thus, if you managed to solve the problem above by generating the phi X174 genome, you simply got lucky: there exist many genomes with exactly the same 10-mers as the phi X174 genome! That is why we accept any genome (with the same set of 10-mers and the same length as phi X174 genome) as a solution of the problem above. An even better approach would be to generate a set of all maximal substrings that are shared by *all* possible genomes with the same set of 10-mers as phi X174 genome. We will discuss this solution later on in this capstone.

EXERCISE BREAK: What is the minimal value of the *k*-mer size for which the phi X174 genome can be uniquely reconstructed from its *k*-mer composition?

Assembling Puzzles from Repetitive Pieces

Real genomes are full of repeats, e.g., over 50% of the human genome is made up of repeats. The most promiscuous repeat in the human genome (about 300 nucleotide-long **Alu sequence**) is repeated (with some small variations) over a million times. An analogy illustrating the difficulty of assembling a genome with many repeats is the **Triazzle** jigsaw puzzle (Figure 6 (left)). People usually put together jigsaw puzzles by connecting matching pieces. However, every piece in the Triazzle matches more than one other piece, e.g., each frog appears several times. If you proceed carelessly, then you will likely match most of the pieces but fail to fit the remaining ones. And yet the Triazzle has only 16 pieces, which should give us pause about assembling a genome from millions of reads.

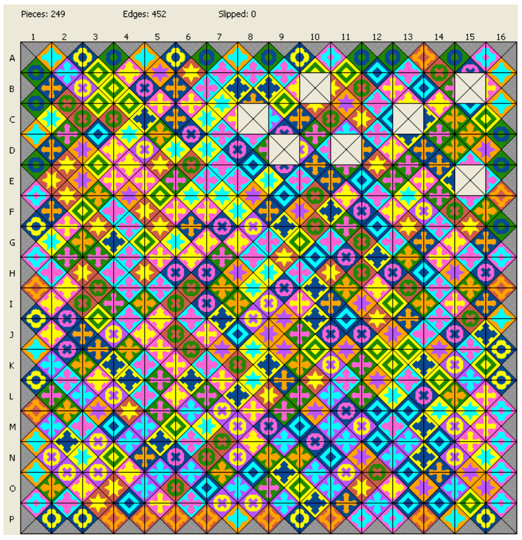


FIGURE 5 (Left) Each Triazzle has only sixteen pieces but carries a warning: "It's Harder than it Looks!" (Right) A nearly Eternity II puzzle with some missing pieces.

EXERCISE BREAK: Design an algorithm for assembling the Triazzle puzzle.

If you got excited solving the problem above, you may try to solve the **Eternity II** puzzle that requires placing 256 square pieces into a 16 by 16 grid (Figure 4 (right)). A $2 million prize was offered for the first solution of this puzzle in 2007, but the competition ended with no solution being found (note seven empty squares in (Figure 4 (right)). The inventor of the Eternity II wrote: "*If you used the world's most powerful computer and let it run from now until the projected end of the universe, it might not stumble across one of the solutions.*" You thus may want to upgrade your laptop before trying to solve it.

Puzzle Assembly. Develop a program for assembling a smaller version of Eternity II puzzle that requires placing 25 square pieces into a 5-by-5 grid.

String Reconstruction as a Hamiltonian Path Problem

Overlap graph for k-mer composition

To apply the concept of the overlap graph for solving the String Reconstruction Problem, we define SUFFIX(*Pattern*) and PREFIX(*Pattern*) as the last and first (*k*-1)-mers in a *k*-mer *Pattern*, respectively. We form a vertex for each *k*-mer in *Patterns* and connect *k*-mers *Pattern* and *Pattern′* by a directed edge from *Pattern* to *Pattern′* if SUFFIX(*Pattern*)=PREFIX(*Pattern′*). The resulting overlap graph is denoted OVERLAP(*Patterns*). See Figure 6.

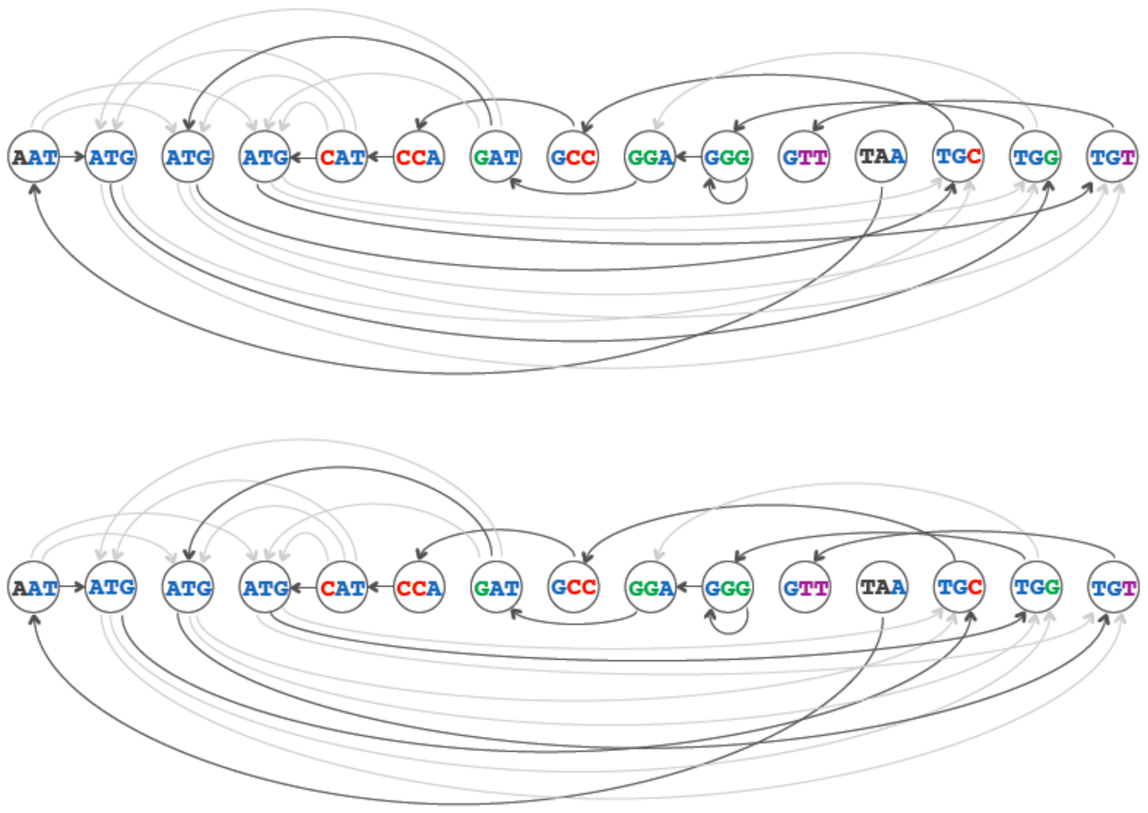


FIGURE 6 (Top) The paths in the overlap graph spelling out TAATGCCATGGGATGTT (top) and TAATGGGATGCCATGTT (bottom). These two strings differ by exchanging the positions of CC and GG but have the same 3-mer composition.

We now know that to solve the String Reconstruction Problem, we are looking for a **Hamiltonian path** that visits every vertex exactly once:

Hamiltonian Path Problem: Construct a Hamiltonian path in a graph.
Input: A directed graph.
Output: A path visiting every vertex in the graph exactly once (if such a path exists).

We do not ask you to develop an efficient algorithm for the Hamiltonian Path Problem since, as you have already learned, the Hamiltonian Path Problem is NP-complete. Instead, we want you to meet Nicolaas de Bruijn, a Dutch mathematician who was interested in the String Reconstruction Problem 70 years ago. You may be wondering how meeting de Bruijn would help you solve the String Reconstruction Problem since, as we just saw, it amounts to an NP-complete Hamiltonian Path Problem! You are about to discover the art of problem formulations: the fact that we reduced the String Reconstruction Problem to the Hamiltonian Path Problem does not necessarily means that it is the right way to attack this problem!

Universal strings

A binary string is ***k*-universal** if it contains every binary *k*-mer exactly once. For example, **0001110100** is a 3-universal string, as it contains each of the eight binary 3-mers (**000**, **001**, **011**, **111**, **110**, **101**, **010**, and **100**) exactly once. Finding a *k*-universal string is equivalent to solving the String Reconstruction Problem when the *k*-mer composition is the collection of all binary *k*-mers. Thus, finding a *k*-universal string is equivalent to finding a Hamiltonian path in the overlap graph formed on all binary *k*-mers (Figure 7).

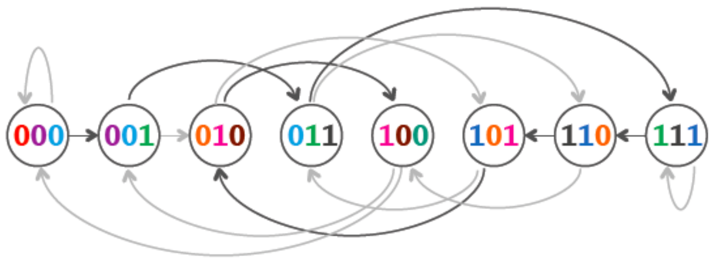


FIGURE 7. A Hamiltonian path highlighted in the overlap graph of all binary 3-mers. This path spells out the 3-universal binary string 0001110100.

EXERCISE BREAK: Construct a 4-universal string. How many different 4-universal strings can you construct?

Although the Hamiltonian path in Figure 7 can be found by hand, de Bruijn was interested in constructing *k*-universal strings for arbitrary values of *k*. For example, to find a 20-universal string, you would have to consider a graph with over a million vertices. It is unclear how to find a Hamiltonian path in such a huge graph, or even whether such a path exists!

De Bruin realized that modeling the search for a universal string as the Hamiltonian Path Problem is a dead end. Instead of searching for Hamiltonian paths in huge graphs, he developed a different (and somewhat non-intuitive) way of representing a *k*-mer composition using a graph.

De Bruijn Graphs

To follow in de Bruijn's footsteps, we will consider the **Eulerian Path Problem** that is seemingly very similar to the Hamiltonian Path Problem. We already discussed Eulerian paths in *undirected* graphs in this Specialization, but now we will consider Eulerian paths in *directed* graphs.

Eulerian Path Problem: Construct an Eulerian path in a directed graph.
Input: A directed graph.
Output: A path visiting every edge in the graph exactly once (if such a path exists).

STOP and Think: Construct a graph in which every *k*-mer corresponds to an edge rather than a vertex and where *k*-universal strings correspond to Eulerian paths. How many vertices this graph will have for *k*=10?

If you have difficulty solving the problem above, here is how de Bruijn solved it. Given a collection of *k*-mers *Patterns*, the vertices of the graph DeBRUIJN**(***Patterns***)** are simply all unique (*k***-**1)-mers occurring as a prefix or suffix of *k*-mers in *Patterns*. For example, say we are given the following collection of 3-mers:

AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT

Then the set of eleven *unique* 2-mers occurring as a prefix or suffix in this collection is as follows:

AA AT CA CC GA GC GG GT TA TG TT

For every *k*-mer in *Patterns*, we connect its prefix vertex to its suffix vertex by a directed edge in order to construct DeBRUIJN(*Patterns*) (Figure 8).

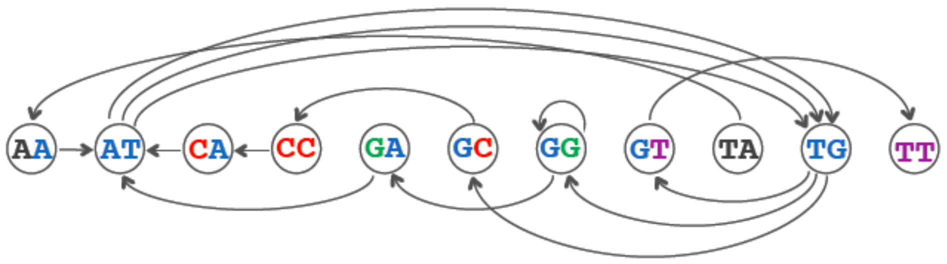


FIGURE 8 The de Bruijn graph of 3-mers: **AAT**, **ATG**, **ATG**, **ATG**, **CAT**, **CCA**, **GAT**, **GCC**, **GGA**, **GGG**, **GTT**, **TAA**, **TGC**, **TGG**, and **TGT**.

Constructing de Bruijn Graph from a set of *k*-mers. Given a set of *k*-mers *Patterns*, construct the de Bruijn graph DeBRUIJN(*Patterns*).

We now have two ways of solving the String Reconstruction Problem. We can either find a Hamiltonian path in the overlap graph or find an Eulerian path in the de Bruijn graph (Figure 9). Note that we have only changed a single word in the statements of the Hamiltonian and Eulerian Path Problems, from finding a path visiting every *vertex* exactly once to finding a path visiting every *edge* exactly once.

STOP and Think: Was it really worth your time to learn two slightly different ways of solving the same problem?

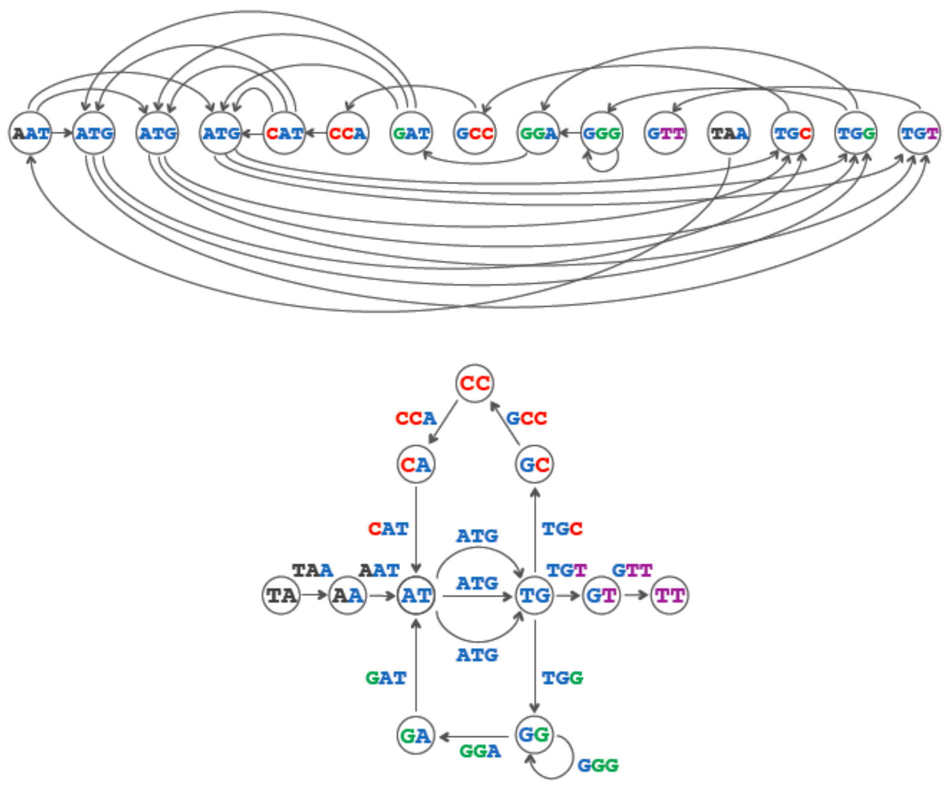


FIGURE 9 The overlap graph (top) and the de Bruijn graph (bottom) for the same set of 3-mers. Note that the de Bruijn graph at the bottom and the de Bruijn graph shown in Figure 8 are the same but differently drawn graphs.

STOP and Think: Which graph would you rather work with, the overlap graph or the de Bruijn graph?

Every reasonable person would probably prefer working with the de Bruijn graph, since it is smaller. However, this would be the wrong reason to choose one graph over the other. In the case of real assembly problems, both graphs will have millions of vertices, and so all that matters is finding an efficient algorithm for reconstructing the genome. To help you make the choice between two approaches, we invite you for a field trip to the 18th Century. Our destination is the Prussian city of Königsberg.

From Bridges of Königsberg to Euler's Theorem

Bridges of Königsberg

Königsberg was comprised of both banks of the Pregel River as well as two river islands; seven bridges connected these four different parts of the city, as illustrated in Figure 10. Königsberg's residents asked a question that became known as the **Bridges of Königsberg Problem**: "Is it possible to set out from my house, cross each bridge exactly once, and return home?"

EXERCISE BREAK: Does the Bridges of Königsberg Problem have a solution?

In 1735, Leonhard Euler drew the graph in Figure 10, which we call *Königsberg*; this graph's vertices represent the four sectors of the city, and its edges represent the seven bridges connecting different sectors. Note that the edges of *Königsberg* are undirected, meaning that they can be traversed in either direction.

STOP and Think: Redefine the Bridges of Königsberg Problem as a question about the graph *Königsberg*.

Eulerian cycles and genome sequencing

We have already defined an Eulerian path as a path traversing each edge of a graph exactly once. A cycle that traverses each edge of a graph exactly once is called an **Eulerian cycle**, and we say that a graph containing such a cycle is **Eulerian**. Note that an Eulerian cycle in Königsberg would immediately provide the residents of the city with the walk they had wanted. We now redefine the Bridges of Königsberg Problem as an undirected version of the Eulerian Cycle Problem:

Eulerian Cycle Problem: Find an Eulerian cycle in a graph.

Input: A directed graph.

Output: An Eulerian cycle in this graph, if one exists.

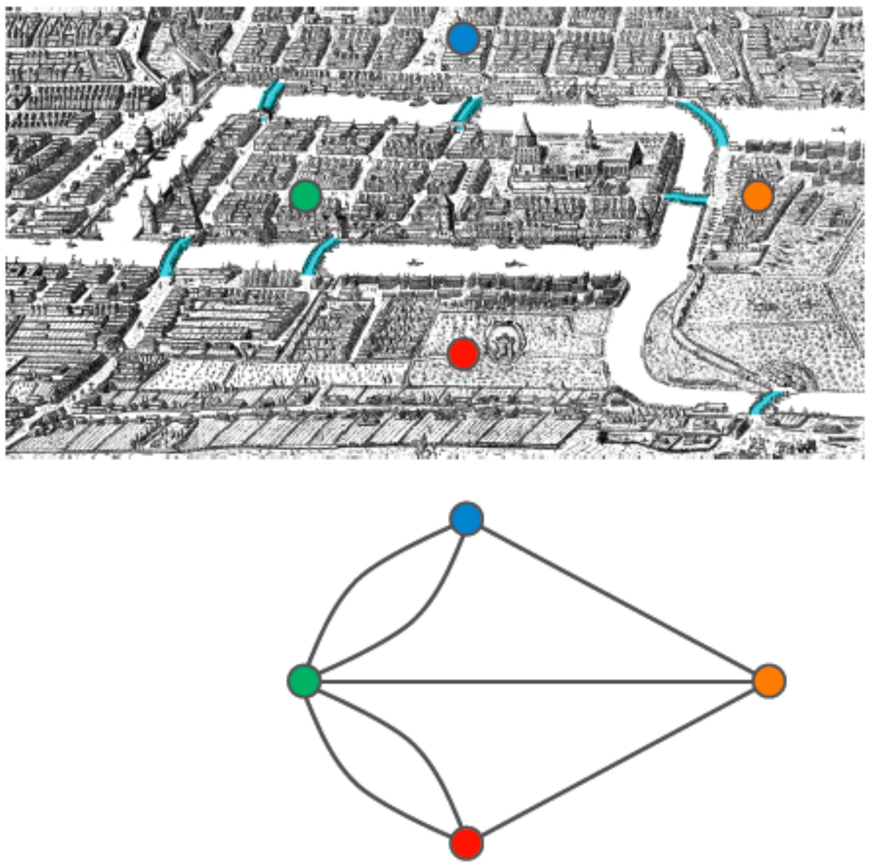


FIGURE 10. (Top) A map of Königsberg, adapted from Joachim Bering's 1613 illustration. The city was made up of four sectors represented by the blue, red, yellow, and green dots. The seven bridges connecting the different parts of the city have been highlighted to make them easier to see. (Bottom) The graph *Königsberg*.

Euler solved the Bridges of Königsberg Problem, showing that no walk can cross each bridge exactly once (i.e., the graph *Königsberg* is not Eulerian), which you may have already figured out for yourself. Yet his real contribution, and the reason why he is viewed as the founder of graph theory, is that he proved a theorem dictating when a graph will have an Eulerian cycle. His theorem immediately implies a polynomial-time algorithm for constructing an Eulerian cycle in any Eulerian graph, even one having millions of edges. Furthermore, this algorithm can easily be extended into an algorithm for constructing an Eulerian path (in a graph having such a path), which will allow us to solve the String Reconstruction Problem by using the de Bruijn graph.

For the first two decades following the invention of DNA sequencing methods, biologists assembled genomes using overlap graphs, since they failed to realize that the Bridges of Königsberg held the key to DNA assembly. It took bioinformaticians some time to figure out that the de Bruijn graph, first constructed to solve the universal string problem, was relevant to genome assembly. Moreover, when de Bruijn graphs were brought to bioinformatics, they were considered an exotic mathematical concept with limited practical applications. Today, de Bruijn graphs have become the dominant approach for genome assembly.

Euler's theorem

Consider an ant walking along the edges of an Eulerian cycle. Every time the ant enters a vertex of this graph by an edge, he is able to leave this vertex by another, unused edge. Thus, in order for a graph to be Eulerian, the number of incoming edges at any vertex must be equal to the number of outgoing edges at that vertex. We define the **indegree** and **outdegree** of a vertex v (denoted IN(v) and OUT(v), respectively) as the number of edges leading into and out of v. A vertex v is **balanced** if IN(v) = OUT(v) , and a graph is balanced if all of its vertices are balanced. Because the ant must always be able to leave a vertex by an unused edge, any Eulerian graph must be balanced.

STOP and Think: We now know that every Eulerian graph is balanced; is every balanced graph Eulerian?

We say that a directed graph is **strongly connected** if it is possible to reach any vertex from every other vertex. Obviously, an Eulerian graph must be both balanced and strongly connected. Euler's Theorem states that each strongly connected and balanced graph is Eulerian. As a result, it implies that we can determine whether a graph is Eulerian without ever having to draw any cycles.

To prove the Euler's theorem, place the ant at any vertex of the graph and let him randomly walk through the graph under the condition that he cannot traverse the same edge twice. If the ant were incredibly lucky— or a genius— then he would traverse each edge exactly once and return back to the initial vertex. However, odds are that he will get stuck somewhere before he can complete an Eulerian cycle, meaning that he reaches a vertex and finds no unused edges leaving that vertex.

STOP and Think: Where is the ant when he gets stuck? Can he get stuck in any vertex of the graph or only in certain vertices?

STOP and Think: After you answer the question above, is there a way to give the ant different instructions so that he selects a longer walk through the graph before he gets stuck? If you can find a longer walk, you will be able to iterate until the Eulerian cycle is found.

For example, imagine that the ant started in the bottom vertex in the graph in Figure 11 (left) and generated a green cycle *Cycle* that has not visited each edge in the graph. Because the graph is strongly connected, some vertices on *Cycle* must have unused edges entering it and leaving it. Naming one of such vertices *v*, we ask the ant to start at v instead of the initial vertex and traverse *Cycle* (thus returning to v), as shown in Figure 11 (middle).

The ant is probably annoyed that we have asked him to travel along the exact same cycle. However, now there are unused edges starting at the vertex v, and so he can continue walking from v after traversing the green cycle, using a new edge each time. The result of his walk is a new cycle in Figure 11 (right), which is larger than the initial *Cycle*.

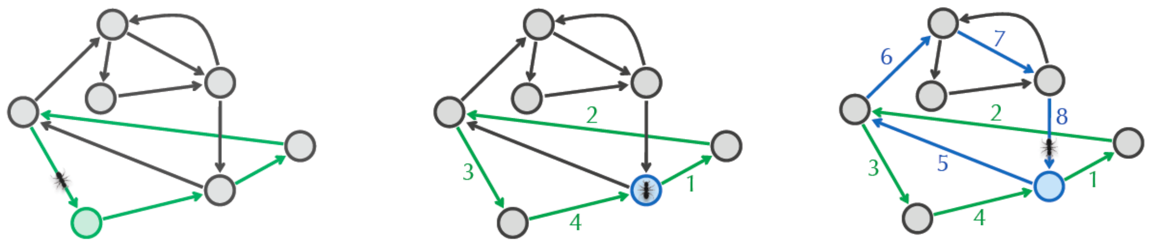


FIGURE 11 (Left) Starting from the bottom vertex, the ant walks along a green cycle *Cycle* (formed by when he gets stuck at the green vertex). In this case, he got stuck before visiting every edge in the graph. (Middle) Starting at a new vertex v (shown in blue), the ant first travels along *Cycle*, returning to v. Note that the blue vertex v, unlike the initial green vertex, has unused outgoing and incoming edges so that the ant can continue walking. (Right) Enlarged cycle consisting of eight edges.

With this hint, you should be ready to give a rigorous proof of the Euler's theorem.

EXERCISE BREAK: Prove Euler's Theorem stating that every balanced, strongly connected directed graph is Eulerian.

EXERCISE BREAK: Formulate and prove an analog of Euler's Theorem for finding Eulerian paths (rather than Eulerian cycles).

Constructing Eulerian cycles

If you proved Euler's Theorem, you may have already come up with an algorithm for constructing an Eulerian cycle. In short, we track the ant's movements until he inevitably produces an Eulerian cycle in a balanced and strongly connected graph, as summarized in the following pseudocode.

```
EulerianCycle(Graph)
form a cycle Cycle by randomly walking in Graph (don't visit the same edge twice!)
while there are unexplored edges in Graph
  select a vertex newStart in Cycle with still unexplored edges
  form a cycle Cycle' by traversing Cycle (starting at newStart) and then randomly walking
  Cycle ← Cycle'
return Cycle
```

It may not be obvious, but a good implementation of **EulerianCycle** will work in linear time. To achieve linear runtime, you would need to use an efficient data structure in order to maintain the current cycle that the ant is building.

Linear-time search for an Eulerian cycle. Implement a linear time algorithm for constructing an Eulerian cycle.

Constructing Universal Strings

Now that you know how to use the de Bruijn graph to solve the String Reconstruction Problem, you can also construct a *k*-universal string for any value of *k*. We should note that de Bruijn was interested in constructing *k*-universal *circular* strings. For example, 00011101 is a 3-universal circular string, as it contains each of the eight binary 3-mers exactly once (Figure 12).

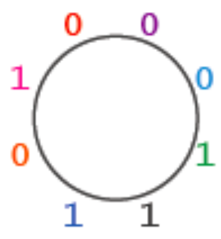


FIGURE 12 The circular 3-universal string 00011101 contains each of the binary 3-mers (000, 001, 011, 111, 110, 101, 010, and 100) exactly once.

***k*-Universal Circular String Problem**: Find a *k*-universal circular string.

Input: An integer *k*.

Output: A *k*-universal circular string.

Like its analogue for linear strings, the *k*-Universal Circular String Problem is just a specific case of a more general problem, which requires us to reconstruct a circular string given its *k*-mer composition. This problem models the assembly of a circular genome containing a single chromosome, like the genomes of most bacteria. We know that we can reconstruct a circular string from its *k*-mer composition by finding an Eulerian cycle in the de Bruijn graph constructed from these *k*-mers. Therefore, we can construct a *k*-universal circular binary string by finding an Eulerian cycle in the de Bruijn graph constructed from the collection of all binary *k*-mers.

EXERCISE BREAK: How many 4-universal circular strings are there?

Even though finding a 20-universal circular string amounts to finding an Eulerian cycle in a graph with over a million edges, we now have a fast polynomial algorithm for solving this problem. Let $BinaryStrings_k$ be the set of all 2^k binary *k*-mers. The only thing we need to do for solving the *k*-Universal Circular String Problem is to find an Eulerian cycle in the graph DeBRUIJN($BinaryStrings_k$), where vertices represent all possible binary **(***k***-**1**)**-mers. Figure 13 illustrates that DeBRUIJN**(***BinaryStrings*$_4$**)** is balanced and strongly connected and is thus Eulerian.

STOP and Think: Prove that for any *k*, DeBRUIJN**(***BinaryStrings*$_k$**)** is Eulerian.

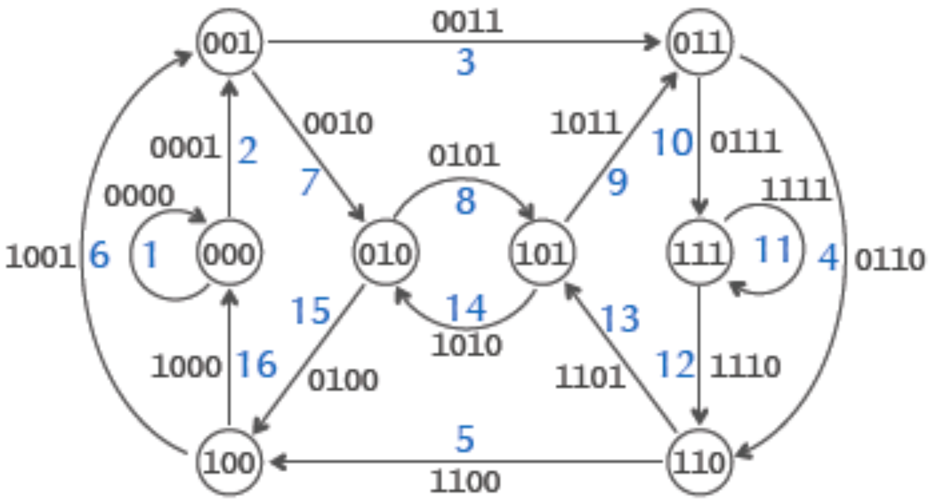


FIGURE 13 An Eulerian cycle spelling the cyclic 4-universal string 0000110010111101 in **DeBRUIJN(***BinaryStrings*$_4$**)**.

Constructing a circular *k*-universal string. Given an integer *k*, construct a *k*-universal string.

Splitting the Genome into Contigs

Given a set of reads from a genome, we define the **coverage of the genome** by these reads as the combined length of all reads divided by the genome length. For example, if the genome has length 5,000 and the combined length of all reads is 100,000, then the coverage is 20X; furthermore, if the reads have length 100, then on average, the probability that there is some read starting at a given position of the genome is 0.2. Since reads are drawn randomly from the genome, and some regions of the genome make it difficult to generate reads (e.g., regions with high frequencies of G and C nucleotides), some positions often have significantly smaller coverage than other positions. To deal with this non-uniformity, we define the **coverage at a given position** of the genome as the number of reads that contain this position. Finally, given a set of reads and a *k*-mer from a genome, we define the **coverage of this *k*-mer** as the number of reads that contain this *k*-mer.

Most assemblies have gaps in *k*-mer coverage, i.e., *k*-mers from the genome that are not present in any reads (remember burnt shreds of paper from the Newspaper Problem?). As the result, since de Bruijn graphs often have missing edges, the search for an Eulerian cycle may fail. For this reason, biologists often settle on assembling **contigs** (long, contiguous segments of the genome) rather than entire chromosomes. For example, a typical bacterial sequencing project may result in 50-100 contigs, ranging in length from a few thousand to a few hundred thousand nucleotides. For many genomes, the order of these contigs along the genome remains unknown. In practice, biologists have no choice but to assemble contigs rather than the entire genome, even in the case of perfect coverage, since repeats prevent them from being able to infer a unique Eulerian cycle. Needless to say, they would prefer to have the entire genome, but the cost of ordering the contigs into a contiguous genome and closing the gaps using more expensive experimental methods is often prohibitive.

Fortunately, we can derive contigs from the de Bruijn graph. A path in a graph is called **non-branching** if IN**(***v***)** = OUT**(***v***)** = 1 for each intermediate vertex *v* of this path, i.e., for each vertex except possibly the starting and ending vertex of a path. A **maximal non-branching path** is a non-branching path that cannot be extended into a longer non-branching path. We are interested in these paths because the strings of nucleotides that they spell out must be present in any genome with a given *k*-mer composition. For this reason, contigs correspond to strings spelled by maximal non-branching paths in the de Bruijn graph. For example, the de Bruijn graph in Figure 14, constructed for the 3-mer composition of the string TAATGCCATGGGATGTT, has nine maximal non-branching paths that spell out the contigs TAAT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG, and GGAT. Biologists often prefer to work with the **condensed de Bruijn** graph, in which each contig is represented by a single edge and edges are labeled by strings of arbitrary length as shown in Figure 15.

Contig Generation Problem: Given a set of *k*-mers *Patterns*, generate all contigs in DeBRUIJN(*Patterns*) and compute the number of contigs for the case when the set *Patterns* consist of all 10-mers from a mutated phi X174 genome.

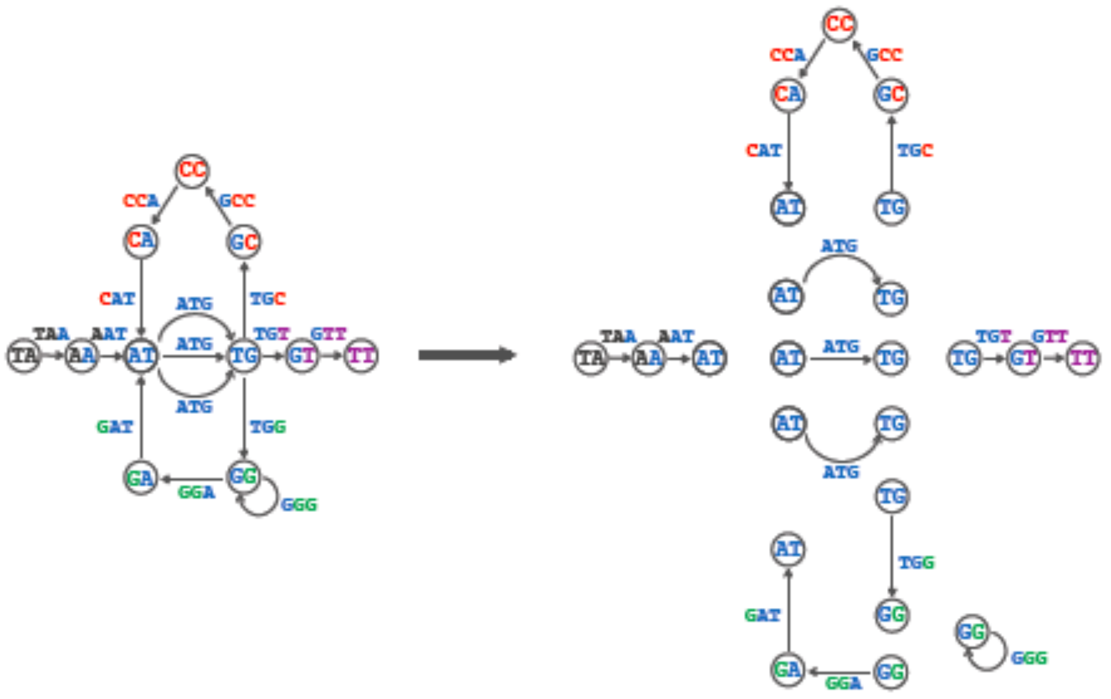


FIGURE 14 Breaking the graph DeBRUIJN$_3$(TAATGCCATGGGATGTT) into nine maximal non-branching paths representing contigs TAAT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG, and GGAT.

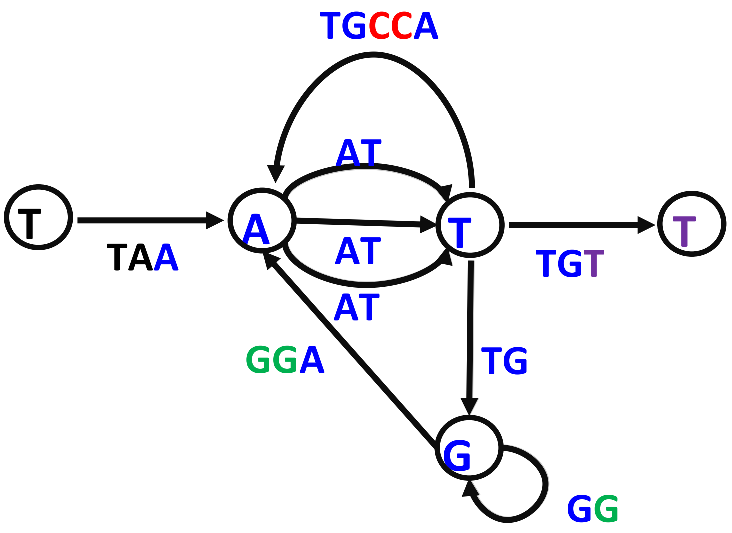


FIGURE 15 Condensed version of the de Bruijn graph from Figure 13 (left).

If you have difficulties finding maximal non-branching paths in a graph, check out the CHARGING STATION: “Maximal Non-Branching Paths in a Graph” in Compeau and Pevzner. *Bioinformatics Algorithms: An Active Learning Approach* (www.bioinformaticsalgorithms.org).

Genome Assembly Faces Real Sequencing Data

Our discussion of genome assembly has thus far relied upon various assumptions. Below, we describe practical challenges introduced by quirks in modern sequencing technologies and some computational techniques that have been devised to address these challenges.

Inferring multiplicities of k-mers

We have now learned how to assemble a genome from its *k*-mer composition, i.e., when information about multiplicity of each repeat is known. In reality, the multiplicities of *k*-mers are not known, but we still can construct the de Bruijn graph that however will be unbalanced in this case. This graph may help you figure out the missing information about multiplicities of edges (*k*-mers) in this graph. For example, if there are three incoming edges (v_1,w), (v_2,w), (v_3,w), in a vertex w and one outgoing edge, chances are that the multiplicity of the outgoing edge is 3. However, if there are 2 incoming edges into vertex v_1, should we assume that the multiplicity of this edge is 4???

EXERCISE BREAK: Given all 10-mers from the phi X174 genome, construct the condensed de Bruijn graph of these 10-mers and infer the multiplicities of each 10-mer that is repeated more than once.

Even if you solved the Exercise Break above, the problem of inferring multiplicities of *k*-mers is not as simple as it may appear (try to solve it for 8-mers rather than 10-mers and you will see why).

STOP and Think: The problem of inferring multiplicities of *k*-mers can be formulated as one of the problems you have already studied in this Specialization. Can you figure out which one?

If we were able to infer multiplicities of all *k*-mers and assign them to edges of the condensed de Bruijn graph, then the total multiplicity of all edges entering a given vertex will be equal to the total multiplicity of all edges leaving this vertex (for a circular genome). Therefore, we will interpret the multiplicity of each edge as a **network flow** along the edge resulting in the balanced flow for each vertex. The problem, however, is that we don't know what the multiplicities are and thus cannot construct this network flow…

Actually, we do, at least for some edges! Although real genomes have many repeats, these repeats are usually short. Thus, we can assume that all *long* edges in the condensed de Bruijn graph have multiplicity 1. As a result, since repeats in phage genomes are usually short, we can safely assume that all edges in the condensed de Bruijn graph of the phi X174 genome corresponding to long contigs (e.g., longer than 1000 nucleotides) have multiplicity 1.

The **circulation problem** is a generalization of the network flow problem, with the added constraint of a lower bound on flows, and with the requirement that the flow is balanced in *all* vertices, including the source and sink. In other words, for each vertex in the network, the total incoming flow is equal to the total outgoing flow. Instead of a single upper bound $high(v,w)$ for the flow on an edge (v,w) as in the standard network flow problem, there is also a lower bound $low(v,w)$, and the flows on edges should satisfy the following constraints:

$$low(v,w) \leq flow(v,w) \leq high(v,w)$$

Finding a flow assignment satisfying these constraints gives a solution to the circulation problem:

Circulation Problem. Reduce the Circulation Problem to the Maximal Flow in a Network Problem, and use the Ford-Fulkerson algorithm to find a circulation. Apply the program you developed to infer multiplicities of all edges in the de Bruijn graph of all 8-mers from the mutated phi X174 genome. What is the maximal multiplicity of 8-mers in the mutated phi X174 genome?

STOP and Think: The Circulation Problem may have multiple solutions. Design an algorithm to figure out whether there exists a unique solution of the Circulation Problem.

Breaking reads into k-mers

We have taken for granted that modern sequencing machine can generate reads that contain all *k*-mers present in the genome, but this assumption of "perfect *k*-mer coverage" does not hold in practice. For example, the popular **Illumina sequencing technology** generates reads that are 250 nucleotides long, but this technology still misses many 250-mers present in the genome, and nearly all the reads that it does generate have sequencing errors.

STOP and Think: Given a genome and a set of error-free reads of length k that form an imperfect k-mer coverage of this genome, is there a value $l < k$ so that the same reads have perfect l-mer coverage? If yes, what is the maximum value of this parameter?

Figure 16 shows four 10-mer reads that capture some, but not all, of the 10-mers in an example genome. However, if we take the counter-intuitive step of breaking these reads into *shorter* 5-mers, then these 5-mers exhibit perfect coverage. This read-breaking approach, in which we break reads into shorter *k*-mers, is used by many modern assemblers.Read-breaking must deal with a practical trade-off. On the one hand, the smaller the value of k, the larger the chance that the k-mer coverage is perfect. On the other hand, smaller values of k result in a more tangled de Bruijn graph, making it difficult to infer the genome from this graph (Figure 17). In the case of reconstructing a circular bacterial genome from its k-mers, our goal is to select a value of *k* that results in a de Bruijn graph represented by a single cycle. However, when we reconstruct a genome from reads and break reads into *k*-mers, it is not even clear whether there exists a value of *k* that results in the de Bruijn graph represented by a single cycle.

```
ATGCCGTATGGACAACGACT
ATGCCGTATG
  GCCGTATGGA
     GTATGGACAA
          GACAACGACT
```

```
ATGCCGTATGGACAACGACT
ATGCC
 TGCCG
  GCCGT
   CCGTA
    CGTAT
     GTATG
      TATGG
       ATGGA
        TGGAC
         GGACA
          GACAA
           ACAAC
            CAACG
             AACGA
              ACGAC
               CGACT
```

FIGURE 16 Breaking 10-mer reads (left) into 5-mers results in perfect coverage of a genome ATGCCGTATGGACAACGACT by 5-mers (right).

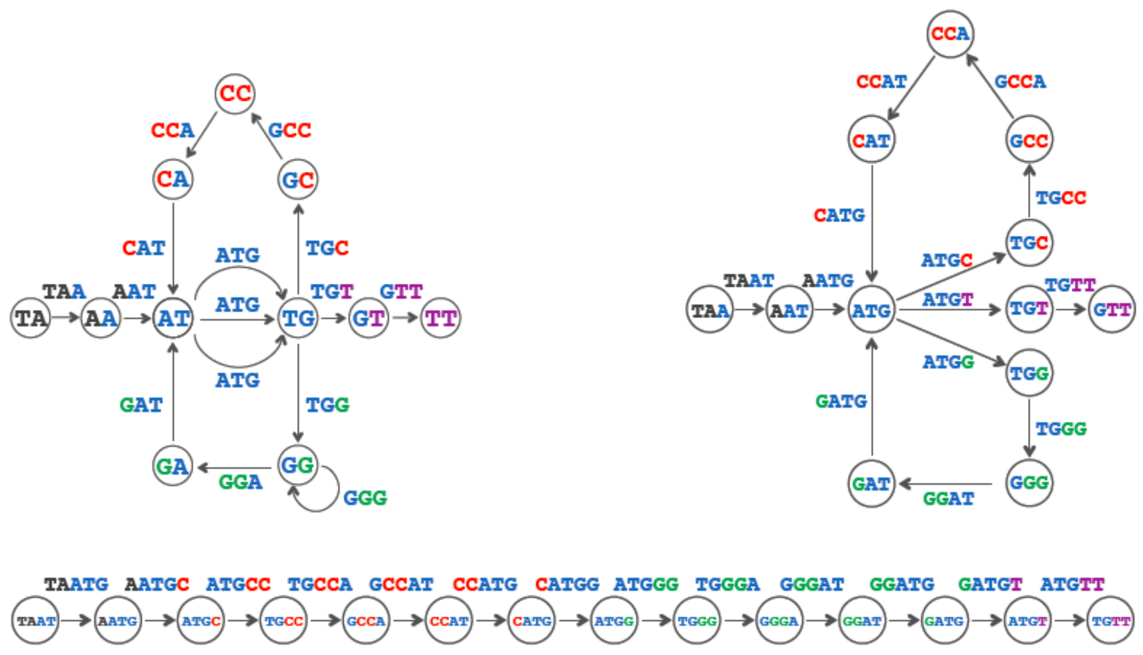


FIGURE 17 The graph DeBRUIJN$_4$(TAATGCCATGGGATGTT) (top right) is less tangled than the DeBRUIJN$_3$(TAATGCCATGGGATGTT) (top left). The graphDeBRUIJN$_5$(TAATGCCATGGGATGTT)

shown at the bottom is a path.

Selecting optimal *k*-mer size for constructing the de Bruijn graph. Given only 300 100-nucleotide long error-free reads from the phi X174 genome, there is a choice of values of *k* for breaking these reads into *k*-mers and further constructing the de Bruijn graph on *k*-mers from reads. Are there "optimal" values of *k* that result in a de Bruijn graph represented by a single cycle? If yes, output minimal and maximum optimal values of *k*.

STOP and Think: Can it happen that there is no single value of *k* that results in a single circular contig but there is nevertheless a smart way to assemble reads in a single circular contig? Is it possible to generalize the concept of the de Bruijn graph so that it is constructed from *k*-mers of *various* sizes rather than the *k*-mers of a *single* fixed size?

Assembling error-prone reads

Error-prone reads represent yet another barrier to real sequencing projects. Adding the single erroneous read CGTACGGACA to the set of reads in Figure 17 (with a single error that misreads T as C) results in erroneous 5-mers CGTAC, GTACG, TACGG, ACGGA, and CGGAC after read breaking. These 5-mers result in an erroneous path from vertex CGTA to vertex GGAC in the de Bruijn graph (Figure 18 (top)), meaning that if the correct read CGTATGGACA is generated as well, then we will have two paths connecting CGTA to GGAC in the de Bruijn graph. This structure is called a **bubble**, which we define as two short disjoint paths (e.g., shorter than some threshold length) connecting the same pair of vertices in the de Bruijn graph.

Bubble Detection Problem. Design an algorithm for detecting bubbles in a directed graph. Output the number of bubbles formed by short disjoint paths (of length less than *t*) in the de Bruijn graph constructed from the *k*-mers occurring in 1000 error-prone reads from a mutated phi X174 genome.

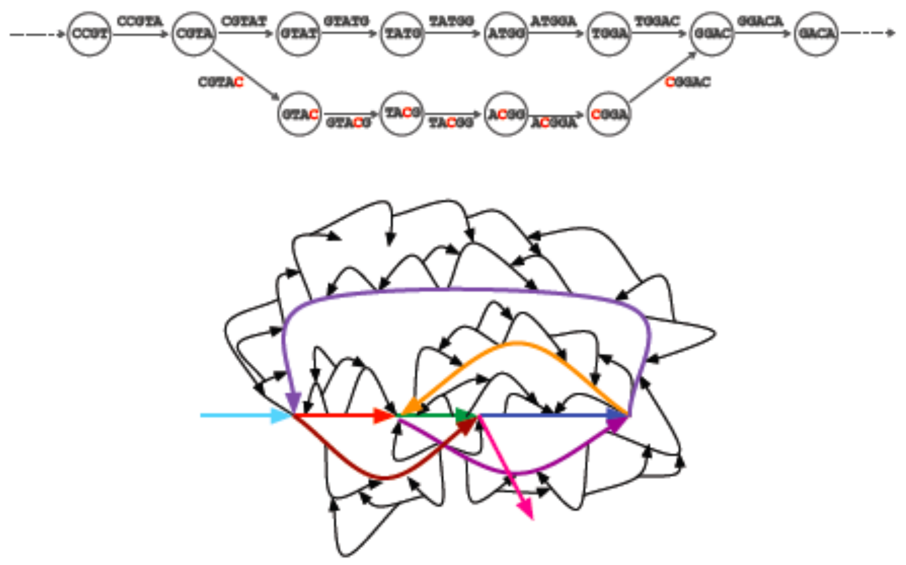


FIGURE 18 (Top) A correct path CGTAèGTATèTATGèATGGèTGGAèGGAC, along with an incorrect path CGTAèGTACèTACGèACGGèCGGAèGGAC resulting from an erroneous read, form a bubble in a de Bruijn graph, making it difficult to identify which path is correct. (Bottom) An illustration of a de Bruijn graph with many bubbles. Bubble removal should leave only the colored paths remaining.

After a bubble is detected, we must decide which of the two alternative paths in the bubble to remove. Since Illumina reads have few errors (the probability of a substitution error at a given base is ≈0.01) and since these errors are rather uniformly distributed along the reads, many existing assemblers remove a path with lower *k*-mer coverage by reads (such paths are often triggered by errors in reads). If each edge (*k*-mer) in the de Bruijn graph is assigned its coverage, then we can compute the average *k*-mer coverage of each path in a bubble and remove the path with smaller coverage (ties are broken arbitrarily).

EXERCISE BREAK: Construct a graph similar to the graph in Figure 18 (top) for an erroneous read CGTATGGACA. Do you see a bubble?

Bubble removal is just one of complications that you will face implementing your own assembler. If you solved the Exercise Break above then you have learned that another difficulty is **tips,** error-prone ends of the reads that do not form a bubble but instead form a path starting in a vertex without incoming edges or ending in a vertex without outgoing edges in the de Bruijn graph. Tips should be removed iteratively because removing a tip can expose another tip.

Tip Removal Problem. Design an algorithm for iterative detection and removal of tips in an arbitrary graph. Output the number of removed edges during the tip removal in the de Bruijn graph constructed from the *k*-mers occurring in 1000 error-prone reads from the mutated phi X174 genome.

Existing assemblers remove bubbles and tips from the de Bruijn graphs. The practical challenge with bubble and tip removal is that, since nearly all reads have errors, de Bruijn graphs have millions of bubbles and tips (Figure 18 (bottom)). Bubble removal occasionally removes the correct path, thus introducing errors rather than fixing them. To make matters worse, in a genome having inexact repeats, where the repeated regions differ by a single nucleotide or some other small variation, reads from the two repeat copies will also generate bubbles in the de Bruijn graph because one of the copies may incorrectly appear to be an erroneous version of the other. Applying bubble removal to these regions introduces assembly errors by making approximate repeats appear more similar than they actually are. Thus, genome assemblers attempt to distinguish bubbles caused by sequencing errors (which should be removed) from bubbles caused by variations in repeats (which should be retained).

Next, we will add a layer of complexity, and you will assemble 1000 *error-prone* reads from a mutated phi X174 genome. For simplicity, we will assume that all of the errors in simulated error-prone reads represent substitutions of nucleotides (i.e., no insertions or deletions).

Assembling phi X174 genome from error-prone reads using de Bruijn graphs. Given 1000 error-prone reads from a mutated phi X174 genome, construct the de Bruijn graph for these reads (for a properly chosen *k*-mer size) and "simplify" it by removing bubbles and tips. Use coverage arguments during the bubble removal process. How many edges are left in the resulting graph after completing the simplification process? How many contigs have you generated and have you made any errors in the resulting contigs during bulge and tip removals? What was the tag inserted in this dataset?

Assembling double-stranded DNA

Before you move towards assembling real DNA sequencing data, we have to acknowledge that, for the sake of simplicity, we have been hiding a "little" detail from you: since DNA consists of two **complementary strands** (Figure 19), reads in real sequencing datasets are drawn from both strands (the phi X174 genome is a rare exception since phi X174 is a single-stranded virus).

In 1953, James Watson and Francis Crick completed their landmark paper on the DNA double helix formed by two complementary DNA strands. Recall from your high school biology course that nucleotides A and T are complements of each other, as are C and G. Figure below shows a strand AGTCGCATAGT and its complementary strand ACTATGCGACT. At this point, you may think that we have made a mistake, since the complementary strand in Figure 19 reads out TCAGCGTATCA from left to right rather than ACTATGCGACT. We did not: DNA strands have directions and the complementary strand runs in the opposite direction to the template strand.

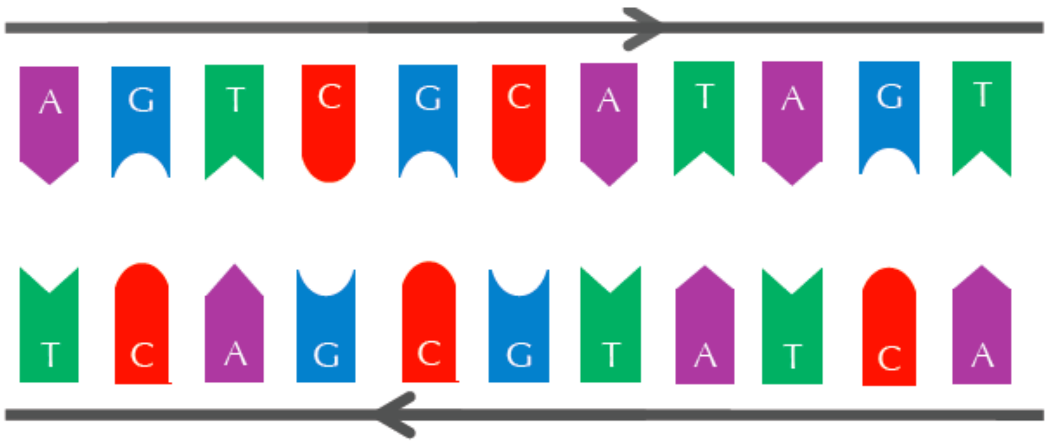


Figure 19 Complementary strands of DNA.

To assemble real genomes, bioinformaticians must handle reads from both DNA strands without knowing in advance which strand each read comes from. To address this challenge, they first add the reverse complement of each read to the set of reads, effectively doubling the number of reads. In an ideal world, the de Bruijn graph formed from all these reads would consist of two (topologically identical but differently labeled) connected components, one for each DNA strand (Figure 20).

In reality, these two components will be "glued" together, since there are many reverse complementary *k*-mers within a single strand in genomes. Indeed, in addition to direct repeats (like ATG in **G**ATG**T**ATG**A**), genomes have many **inverted repeats,** in which one substring is the reverse complement of another (like ATG/CAT in **G**ATG**T**CAT**A**). As a result, while the single strand **G**ATG**T**CAT**A** has no repeated 3-mers, making its assembly trivial, the reverse complementary strands **G**ATG**T**CAT**A** and **T**ATG**A**CAT**C** have repeated 3-mers. Figure 20 illustrates that inverted repeats complicate the assembly.

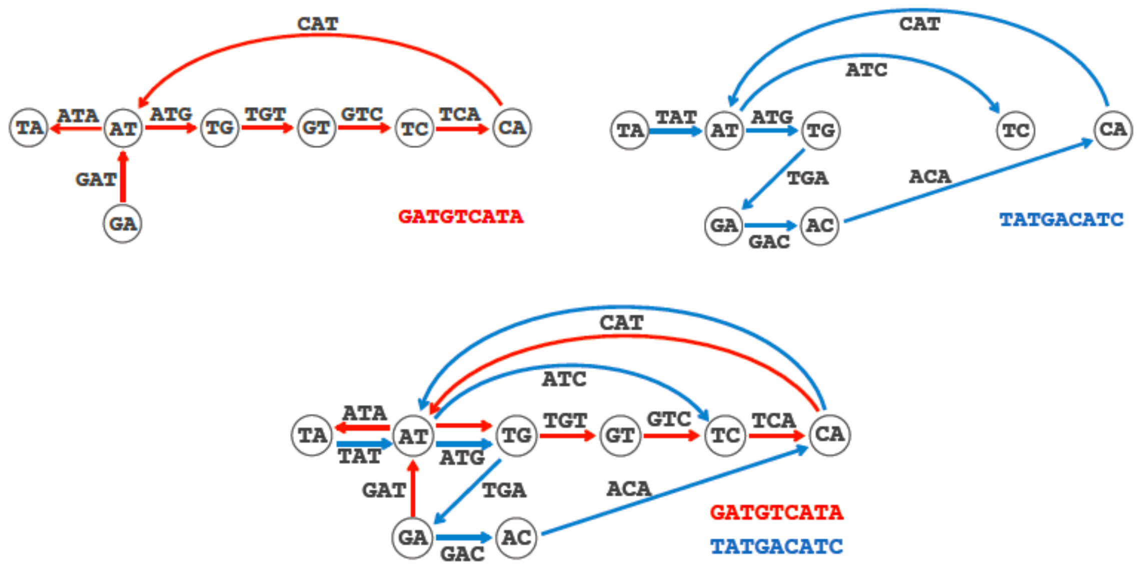


FIGURE 20 (Top) Reconstructing genomes GATGTCATA and TATGACATC from their corresponding graphs DEBRUIJN$_3$(GATGTCATA) and DEBRUIJN$_3$(TATGACATC) can be done easily, as there is only one way to traverse each graph. (Bottom) The de Bruijn graph formed by combining 3-mers from GATGTCATA and its reverse complement TATGACATC. Reconstructing the original genome is now a nontrivial problem.

How good is your assembly?

We will evaluate the quality of assemblies using **QUAST**, **Qu**ality **As**sessment **T**ool for Genome Assembly (Gurevich et al., 2013). QUAST works in two modes (depending on whether the reference genome is known or unknown) and computes various metrics (see FAQ: What Are Assembly Quality Metrics?). Note that most metrics output by QUAST ignore short contigs and are based on long contigs only (e.g., longer than 1000 nucleotides). Biologists are mainly interested in long contigs because short contigs often contain only fragments of genes rather than entire genes (the average length of genes in the *E. coli* genome is approximately 800 nucleotides).

To use QUAST, you will need to generate the "contigs.fasta" file, which defines the contigs resulting from your assembly. See FAQ: "Data Formats for Representing Biological Sequences" to learn how to represent the output of your assembler. You can either download QUAST or use its online version at http://quast.bioinf.spbau.ru/.

You should now understand the practical considerations involved in genome sequencing and be ready to move from sequencing of short viruses to sequencing bacterial genomes that typically contain millions of nucleotides

Assembling the Smallest Bacterial Genome

Nasuia deltocephalinicola is a bacterium that lives symbiotically inside leafhoppers. Its sheltered life has allowed it to reduce its genome to only about 112,091 nucleotides. With only 137 genes, it lacks some genes necessary for survival: since the insects gave them a welcoming home, the bacteria cast aside many essential genes, such as the genes responsible for energy generation (Bennet and Moran, 2013).

The leafhoppers are a nightmare for farmers, causing damage to many vegetables; yet they would be helpless without their bacterial friends. *N. deltocephalinicola* carry out chemical reactions on the sap (that leafhoppers eat) to build amino acids, which the leafhoppers assemble into proteins. Once the ancestors of *N. deltocephalinicola* infiltrated insects, they were able to lose DNA without paying a price.

In fact, *N. deltocephalinicola* has such a small genome that biologists have conjectured that it is losing its "bacterial" identity and turning into a part of the host's genome. This transition from bacterium to a part of the host genome has happened many times during evolutionary history, e.g., the mitochondrion responsible for energy production in human cells was once a free-roaming bacterium that we assimilated in the distant past.

Assembling *N. deltocephalinicola* genome from error-free reads. Given a set of simulated 100-nucleotide long error-free reads from a mutated *N. deltocephalinicola* genome, use the de Bruijn graph approach to reconstruct the *N. deltocephalinicola* genome. How many contigs does your reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling *N. deltocephalinicola* genome from error-prone reads. Given a set of simulated error-prone reads from a mutated *N. deltocephalinicola* genome, use the de Bruijn graph approach to reconstruct the *N. deltocephalinicola* genome. How many contigs does your reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling *N. deltocephalinicola* genome from real reads. Given a set of real reads from (double-stranded) *N. deltocephalinicola* genome, use the de Bruijn graph approach to reconstruct the *N. deltocephalinicola* genome. How many contigs does your reconstruction have? How many errors (if any) does your reconstruction have? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling the *E. coli* X Genome

You are now ready to assemble the genome of the *E. coli* X strain that caused the European outbreak in 2011. We will start from assembling simulated reads and later assemble real reads that we already uploaded to BaseSpace cloud platform, which you can access at https://basespace.illumina.com/s/vozgiZibcX79.

Assembling *E. coli* X genome from simulated error-free reads. Given a set of simulated error-free reads from a mutated *E. coli* X genome, use the de Bruijn graph approach to reconstruct the *E. coli* X genome. How many contigs does your reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling *E. coli* X genome from simulated error-prone reads. Given a set of simulated error-prone reads from a mutated *E. coli* X, use the de Bruijn graph approach to reconstruct the *E. coli* X genome. How many contigs does your reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling *E. coli* X genome from real reads. Given a set of real reads from *E. coli*

X (from 16-year old girl in Hamburg), use the de Bruijn graph approach to reconstruct the *E. coli* X genome. How many contigs does your reconstruction have? How many errors (if any) does your reconstruction have? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling Genomes from Read-Pairs

From reads to read-pairs

Previously, we described an idealized form of genome assembly in order to build up your intuition about de Bruijn graphs. We will now discuss a practically motivated topic that will help you appreciate the advanced methods used by modern assemblers.

De Bruijn graphs become less and less tangled when read length increases (Figure 17). As soon as read length exceeds the length of all repeats in a genome (provided the reads have no errors), the de Bruijn graph turns into a cycle (for reads from a circular bacterial genome under the condition of perfect *k*-mer coverage). However, despite many attempts, biologists have not yet figured out how to generate long and accurate reads. The most accurate sequencing technology available today generate reads that are only about 300 nucleotides long, which is too short to span most repeats, even in short bacterial genomes.

We saw earlier that the string TAATGCCATGGGATGTT cannot be uniquely reconstructed from its 3-mer composition since there exists another string with the same 3-mer composition (TAATGGGATGCCATGTT).

Increasing read length would help identify the correct assembly, but since increasing read length presents a difficult experimental problem, biologists have suggested an indirect way of increasing read length by generating **read-pairs**, which are pairs of reads separated by a fixed distance *d* in the genome (Figure 21). A simple but inferior way to assemble these read-pairs is to ignore the paired information and to construct the de Bruijn graph of individual reads (3-mers) within the read-pairs.

STOP and Think: Read-pairs contain more information than single reads. Can you suggest a way to utilize the pairing information?

You can think about a read-pair as a long "gapped" read of length *k*+*d*+*k*, whose first and last *k*-mers are known but whose middle segment of length *d* is unknown. Nevertheless, read-pairs contain more information than *k*-mers alone, so we should be able to use them to improve our assemblies. If you could infer the nucleotides in the middle segment of a read-pair, you would immediately increase the read length from *k* to 2·*k*+*d*.

AAT-CCA   ATG-CAT   ATG-GAT   CAT-GGA   CCA-GGG   GCC-TGG
GGA-GTT   GGG-TGT   TAA-GCC   TGC-ATG   TGG-ATG

FIGURE 21 Read-pairs sampled from TAATGCCATGGGATGTT and formed by reads of length 3 separated by a gap of length 1.

Transforming read-pairs into long virtual reads

Let *Reads* be the collection of 2*N* reads of length *k* taken from *N* read-pairs. Note that a read-pair formed by *k*-mer reads $Read_1$ and $Read_2$, corresponds to two edges in the de Bruijn graph DeBRUIJN_k(*Reads*). Since these reads are separated by distance *d* in the genome, there must be a path of length *k+d+1* in DeBRUIJN_k(*Reads*) connecting the node at the beginning of the edge corresponding to $Read_1$ with the vertex at the end of the edge corresponding to $Read_2$, as shown in Figure 22. If there is only one path of length *k+d+1* connecting these vertices, or if all such paths spell out the same string, then we can transform a read-pair formed by reads $Read_1$ and $Read_2$ into a virtual read of length $2 \cdot k+d$ that starts as $Read_1$, spells out this path, and ends with $Read_2$.

For example, consider the de Bruijn graph in Figure 22, which is generated from all reads present in the read-pairs in Figure 21. There is a unique string spelled by paths of length *k+d+1*=5 between edges labeled **AAT** and **CCA** within a read-pair represented by the gapped read **AAT-CCA**. Thus, from two short reads of length *k*, we have generated a long virtual read of length $2 \cdot k+d$, achieving computationally what researchers still cannot achieve experimentally! After preprocessing the de Bruijn graph to produce long virtual reads, we can simply construct the de Bruijn graph from these long reads and use it for genome assembly.

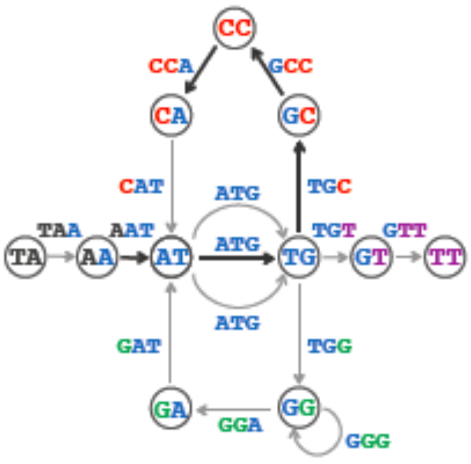


FIGURE 22 The highlighted path of length $k+d+1=3+1+1=5$ between the edges labeled **AAT** and **CCA** spells out **AATGCCA**. There are three such paths because there are three possible choices of edges labeled **ATG**. Thus, the gapped read **AAT-CCA** can be transformed into a long virtual read **AATGCCA**.

Although the idea of transforming read-pairs into long virtual reads is used in many assemblers, we have made an optimistic assumption: "*If there is only one path of length* *k+d+1 connecting these vertices, or if all such paths spell out the same string...*". In practice, this assumption limits the application of the long virtual read approach to assembling read-pairs because highly repetitive genomic regions often contain multiple paths of the same length between two edges, and these paths may spell different strings (Figure 23).

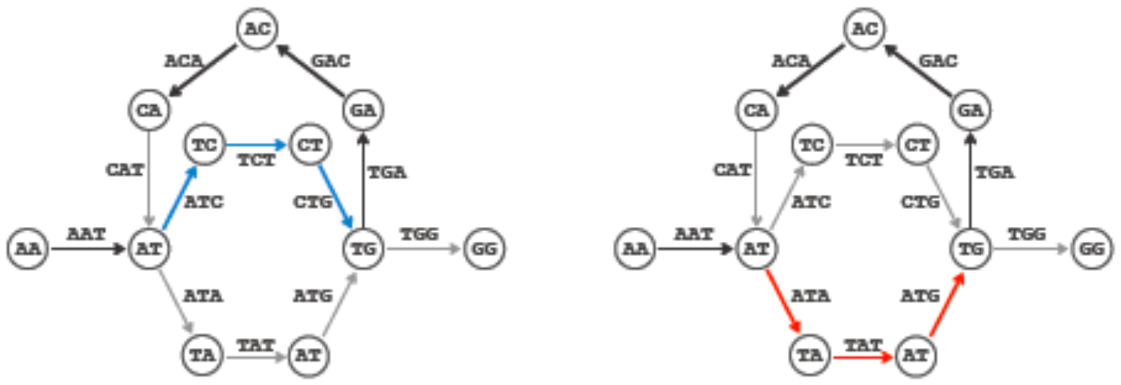


FIGURE 23 (Left) The highlighted path in DeBRUIJN$_3$(**AATCTGACATATGG**) spells out the long virtual read **AATCTGACA**, which is a substring of **AATCTGACATATGG**. (Right) The highlighted path in the same graph spells out the long virtual read **AATATGACA**, which does not occur in **AATCTGACATATGG**.

From composition to paired composition

Given a string *Text*, a **(*k,d*)-mer** is a pair of *k*-mers in *Text* separated by distance *d*. We use the notation (*Pattern*$_1$ | *Pattern*$_2$) to refer to a (*k,d*)-mer whose *k*-mers are *Pattern*$_1$ and *Pattern*$_2$. E.g., (ATG | GGG) is a (3,4)-mer in **TA**ATGCCATGGGATGTT. The ***(k,d)*-mer composition** of *Text*, denoted PairedCOMPOSITION$_{k,d}$(Text), is the set of all (*k,d*)-mers in *Text* (including repeated (*k,d*)-mers). Here are all (3,4)-mers in PairedCOMPOSITION$_{3,1}$(**TA**ATGCCATGGGATGTT):

```
TAA GCC
 AAT CCA
  ATG CAT
   TGC ATG
    GCC TGG
     CCA GGG
```

```
      CAT GGA
       ATG GAT
        TGG ATG
         GGG TGT
          GGA GTT
TAATGCCATGGGATGTT
```

Since the order of (3,1)-mers in PairedCOMPOSITION(TAATGCCATGGGATGTT) is unknown, we list them according to the lexicographic order of the 6-mers formed by their concatenated 3-mers:

```
(AAT|CCA)  (ATG|CAT)  (ATG|GAT)  (CAT|GGA)  (CCA|GGG)  (GCC|TGG)
(GGA|GTT)  (GGG|TGT)  (TAA|GCC)  (TGC|ATG)  (TGG|ATG)
```

Note that, although there are repeated 3-mers in the 3-mer composition of this string, there are no repeated (3,1)-mers in its paired composition. Furthermore, although TAATGCCATGGGATGTT and TAAATGCCATGGGATGTT have the same 3-mer composition, they have different (3,1)-mer compositions. Thus, if we can generate the (3,1)-mer composition of these strings, then we will be able to distinguish between them. But how can we reconstruct a string from its (*k*,*d*)-mer composition? And can we adapt the de Bruijn graph approach for this purpose?

String Reconstruction from Read-Pairs Problem: Reconstruct a string from its paired composition.
Input: A collection of paired *k*-mers *PairedReads* and an integer *d*.
Output: A string *Text* with (*k*,*d*) -mer composition equal to *PairedReads* (if such a string exists).

String Reconstruction from Read-Pairs Problem. Develop an algorithm for solving the String Reconstruction from Read-Pairs Problem.

If you have difficulties reconstructing strings from read-pairs, you can learn about the concept of the **paired de Bruijn graph** and read the CHARGING STATION: “Reconstructing a string spelled by a path in the paired de Bruijn graph” in Compeau and Pevzner. *Bioinformatics Algorithms: An Active Learning Approach* (www.bioinformaticsalgorithms.org).

Assembling the *E. coli* X genome from read-pairs

You are now ready to assemble the *E. coli* X genome from read-pairs. We will start from simulated error-free read-pairs with exact distances between reads within each read-pair and gradually increase the complexity so that, in the end, you assemble the real datasets of read-pairs (seeR: Accessing reads from the TY2482 Dataset).

Generation of Illumina reads was a revolution in DNA sequencing technologies. If you want to learn more about this technology, read FAQs: “How Are Illumina Reads Generated?”and “How Are the Read-Pairs Generated?” Warning: Illumina sequencing technology is rather complex and it may be difficult to understand for people who forgot the basics of molecular biology. Thus, reading the FAQs mentioned above is optional – you should be able to solve the problems below without reading them.

Assembling the *E. coli* X genome from simulated error-free read-pairs with exact distances. Given a set of simulated error-free read-pairs from a mutated *E. coli* X, use the de Bruijn graph approach to reconstruct the *E. coli* X genome. Each read-pair consists from two 100-nucleotide long reads whose starts are separated by exact distance 350 nucleotides. How many contigs does you reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling the *E. coli* X genome from simulated error-free read-pairs with inexact distances. Given a set of simulated error-free read-pairs from a mutated *E. coli* X, use the de Bruijn graph approach to reconstruct the *E. coli* X genome. Each read-pair consists from two 100-nucleotide long reads whose starts are separated by distances varying from 300 to 400 nucleotides. How many contigs does you reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling the *E. coli* X genome from simulated error-prone read-pairs with inexact distances. Given a set of simulated error-prone read-pairs from a mutated *E. coli* X, use the de Bruijn graph approach to reconstruct the *E. coli* X genome. Each read-pair consists from two 100-nucleotide long reads whose starts are separated by distance varying from 300 to 400 nucleotides. How many contigs does you reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report.

Assembling the *E. coli* X genome from real read-pairs. Given a set of real read-pairs from *E. coli* X (TY2482 dataset from 16-year old girl from Hamburg), use the de Bruijn graph approach to reconstruct the *E. coli* X genome. How many contigs does you reconstruction have? How many errors (if any) does your reconstruction have and what was the inserted tag? What is NGA50? Use QUAST to generate the assembly quality report. Don't forget that, in contrast to previous problems, reads in the TY2482 dataset are generated from both DNA strands.

FAQs

What Are Assembly Metrics?

Below we list some metrics for analyzing assembly quality.

N50 statistics: N50 is defined as the maximal contig length for which all contigs greater than or equal to that length comprise at least half of the sum of the lengths of all the contigs. For example, consider five toy contigs with the following lengths: [10, 20, 30, 60, 70]. Here, the total length of contigs is 190, and contigs of length 60 and 70 of total length 130 account for at least half of the total length of contigs. Since the contig of length 70 does not account for half of the total contig length, N50 is equal to 60.

NG50 statistic: The NG50 is a modified version of the N50 statistics that is defined when the length of the genome that is being reconstructed is known. It is defined as the maximal contig length for which all contigs of at least that length comprise at least half of the length of the genome. NG50 enables comparisons between different assemblies for the same genome. For example, consider five toy contigs: [10, 20, 30, 60, 70]. These contigs only add up to 190 nucleotides in length, but say that we know that the genome from which they have been generated has length 300. In this example, the contigs of length 30, 60, and 70 of total length 160 account for at least half of the genome length; but the contigs of length 60 and 70 of total length 130 no longer account for at least half of the genome length. Thus, NG50 is equal to 30.

NGA50 statistic: If we know the reference genome for a species, then we can test the accuracy of a newly assembled contigs against this reference. The NGA50 is a modified version of the NG50 statistics accounting for assembly errors (called **misassemblies**). To compute NGA50, errors in the contigs are accounted for by comparing contigs to a reference genome. All of the misassembled contigs are broken at **misassembly breakpoints**, resulting in a larger number of contigs with the same total length. For example, if there is a misassembly breakpoint at position 10 in a contig of length 30, this contig will be broken into contigs of length 10 and 20.

NGA50 is calculated as the NG50 statistic for the set of contigs resulting from breaking at misassembly breakpoints. For example, consider our previous example, for which the genome length is 300. If the largest contig in [10, 20, 30, 60, 70] is broken into two contigs of length 20 and 50 (resulting in the set of contigs [10, 20, 20, 30, 50, 60]), then contigs of length 20, 30, 50, and 60 of total length 160 account for at least half of the genome length. However, contigs of length 30, 50, and 60 of total length 140 do not account for at least half of the genome length. Thus, NGA50 is equal to 20.

Data Formats for Representing Biological Sequences

Fasta (.fasta, .fa): Text-based format for representing nucleotides or amino acids (the sequence names and comments precede the sequences). Below is an example for an amino acid sequence:

```
>gi|31563518|ref|NP_852610.1| microtubule-associated proteins
1A/1B [Homo sapiens]
MKMRFFSSPCGKAAVDPADRCKEVQQIRDQHPSKIPVIIERYKGEKQLPVLDKTKFLVPDHV
NMSELVKIIRRRLQLNPTQAFFLLVNQHSMVSVSTPIADIYEQEKDEDGFLYMVYASQETFG
FIRENE
```

See an example at https://en.wikipedia.org/wiki/FASTA_format

FastQ (.fastq, .fq). This format is similar to Fasta but also containing information about the quality of each base as letters in the ASCII table (see FAQ: What are the Quality Scores?).

```
@SEQ_ID

GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT

+

!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

See an example at https://en.wikipedia.org/wiki/FASTQ_format

GenBank Format (.gbk): This is a plain-text format for storing sequences and their annotations. The header of the file contains information describing the sequence, such as its type, length, and source. Features of the genome sequence follow the header, and the last element is the DNA sequence, which ends with (and must include) a double slash.

See an example at http://www.ncbi.nlm.nih.gov/genbank/samplerecord/

What are the quality scores?

Unfortunately, sequencing machines are not perfect, and each base in a given read can be erroneous. Fortunately, sequencing machines can assess the probability of this error, providing **Phred quality scores** that specify the accuracy of each base in a read. Phred quality scores are important for deciding whether a sequencing read needs to be filtered or trimmed before assembly.

Phred quality scores are defined as Phred(*base*) = -10 $\log_{10}$ Pr(*base*), where Pr(*base*) is the estimated probability that a given nucleotide *base* in a read is incorrect. For example, if Phred assigns a quality score of 30 to a nucleotide, the probability that this nucleotide is called incorrectly is 0.001. In other words, the larger the quality score, the more reliable the data.

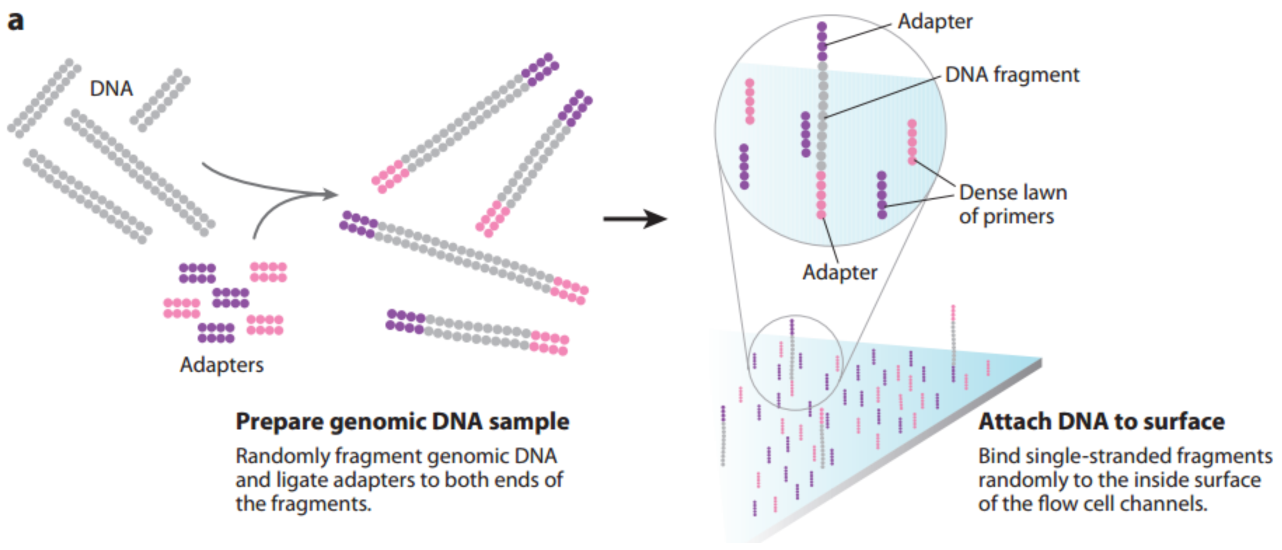
Accessing Reads from the TY2482 Dataset

To access the TY2482 dataset for assembling *E. coli* X genome, open the BaseSpace project https://basespace.illumina.com/s/vozgiZibcX79 and select the "Samples" tab in the left menu. You will see three entries. Each of these entries corresponds to a separate **paired library** – the set of paired DNA reads from a genome with varying insert sizes. In modern DNA sequencing projects, the length of a read-pair is called the **insert size**, which is the total length of both reads plus the length of the gap. The **forward read** in a read-pair is generated from one (forward) strand, whereas the **reversed read** is generated from the complementary strand. Whereas most paired libraries generate reads with insert sizes below 1 kb, biologists often generate libraries with longer (2 kb - 10 kb) insert sizes to improve the assembly. In this capstone project, you will only use the SRR292678 dataset of read-pairs with insert size ≈470 bp and the "forward-reverse" orientation.

How Are Illumina Reads Generated?

In 1998, Shankar Balasubramanian and David Klenerman from Cambridge University developed a novel DNA sequencing method and founded company Solexa. This company was later acquired by San-Diego based company Illumina, and nowadays it is the most popular sequencing technology in the world. It allows biologists to generate billions of reads from a single sample, which allows us to assemble even the largest genomes.

The main idea of the method is simple – during the DNA replication process, we can add nucleotides with different fluorescent labels, and we can detect emitted light to determine which nucleotide was attached. The three figures specify this workflow in more details:



a
DNA
Adapters
Prepare genomic DNA sample
Randomly fragment genomic DNA and ligate adapters to both ends of the fragments.
Adapter
DNA fragment
Dense lawn of primers
Adapter
Attach DNA to surface
Bind single-stranded fragments randomly to the inside surface of the flow cell channels.

Figure S1 (Left) The double-stranded DNA gets chopped up into smaller fragments that are modified by adding **adapters** - short synthetic fragments of DNA that act as anchors for the next step when they hybridize with primers. (Right) The modified DNA fragments are immobilized on the surface of a chip called a **flow cell**, covered by **primers** – short sequences, complementary to the adapters.

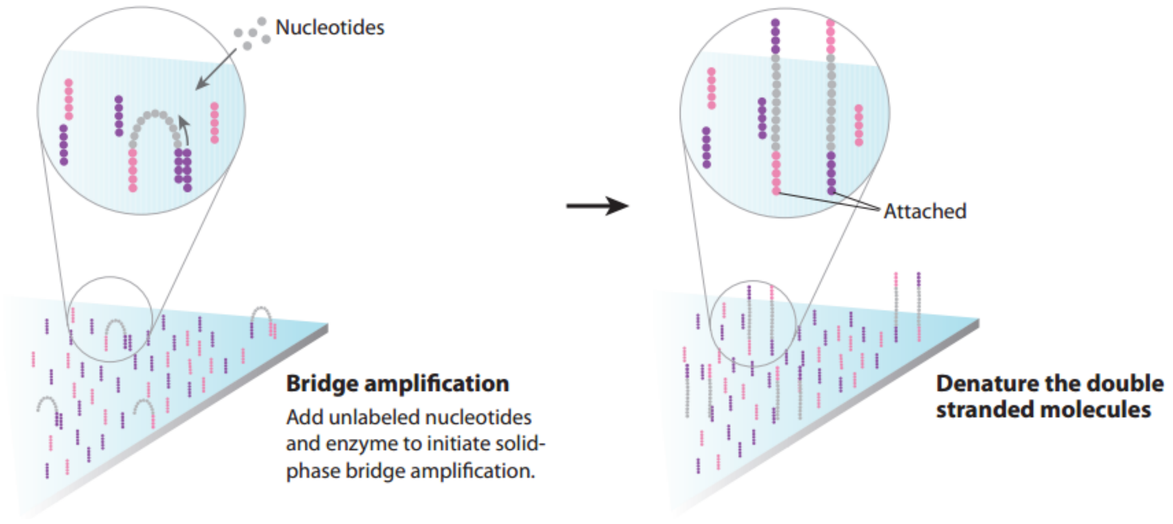

Nucleotides
Attached
Bridge amplification
Add unlabeled nucleotides and enzyme to initiate solid-phase bridge amplification.
Denature the double stranded molecules

Figure S2. (Left) Since the instrument cannot detect luminescence from a single molecule, we need to get multiple copies of the each DNA fragment in close proximity to the original, located in a particular place on the flow cell surface. To achieve this goal the immobilized fragment will bend and attach to the primers nearby, so we can just add nucleotides and **DNA polymerase** for copying DNA, and DNA replication will begin. It looks like a bridge, so this process called **bridge amplification**. (Right) Newly-synthesized copies also can attach to close primers, resulting in a **cluster** on the flow cell surface, composed of the double-stranded copies of the single molecule. Then we can cut one end of the "bridges" using molecular scissors called **restriction enzymes**, and all the reverse strands will be washed off the flow cell, leaving only attached forward strands. It occurs across the whole flow cell, and we got millions of these clusters.

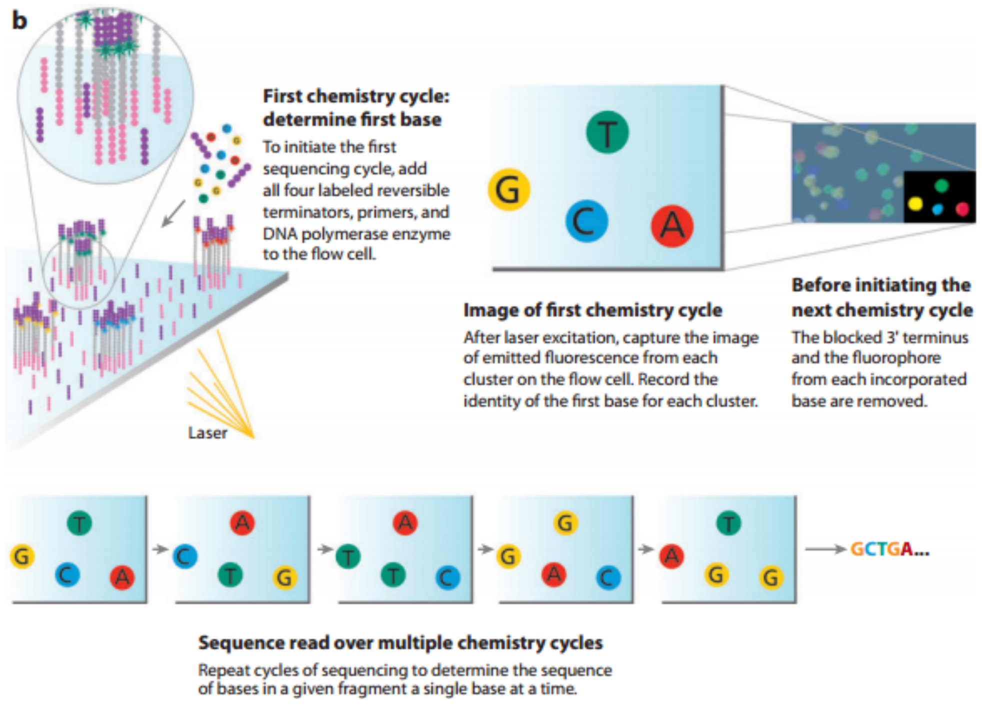
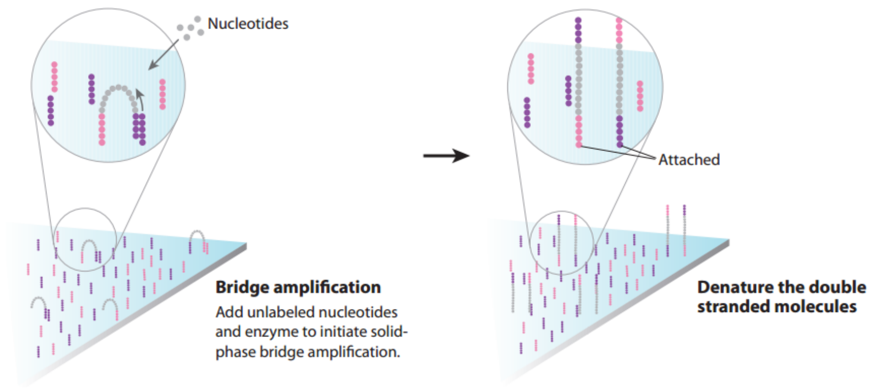

b
First chemistry cycle: determine first base
To initiate the first sequencing cycle, add all four labeled reversible terminators, primers, and DNA polymerase enzyme to the flow cell.
T
G
C
A
Image of first chemistry cycle
After laser excitation, capture the image of emitted fluorescence from each cluster on the flow cell. Record the identity of the first base for each cluster.
Before initiating the next chemistry cycle
The blocked 3' terminus and the fluorophore from each incorporated base are removed.
Laser
GCTGA...
Sequence read over multiple chemistry cycles
Repeat cycles of sequencing to determine the sequence of bases in a given fragment a single base at a time.

Figure S3. Afterwards, sequencing starts. Primers are added, and fluorescently-tagged nucleotides incorporate into the DNA strand starting from the outer end, one at a time. Following the incorporation step, the unused nucleotides and DNA polymerase molecules are washed away. After attaching each nucleotide, a camera takes a picture of the flow cell. Since each of four bases emits its own color after attaching to a DNA, we know which nucleotides are being incorporated at each spot in each cycle. As the nucleotides that are being incorporated in synthesis move from the outer end of the strand towards the flow cell surface, the light intensity fades, so the signal quality decays. That is why we cannot read whole DNA fragments, and we need to overcome it using the **paired-reads**

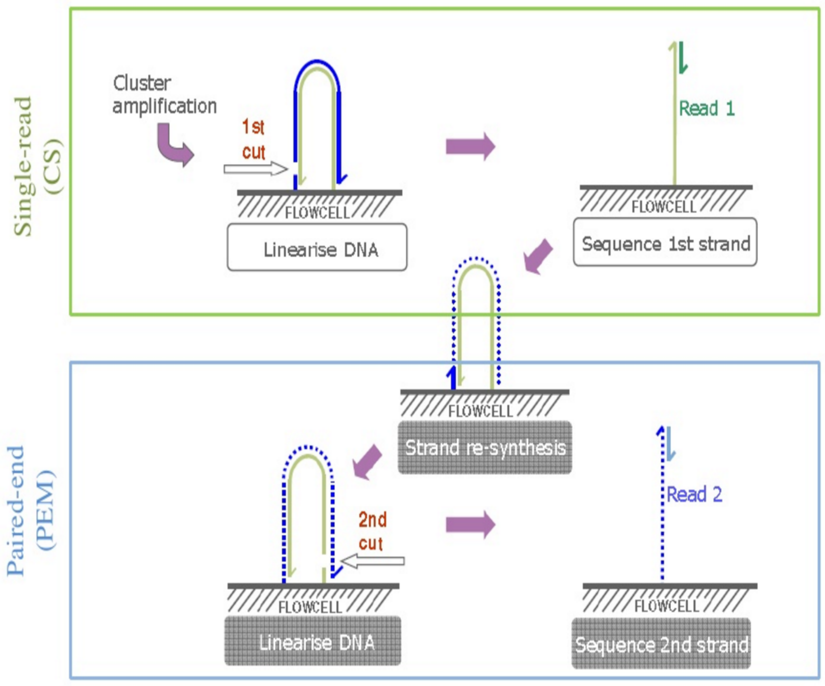

Nucleotides
Attached
Bridge amplification
Add unlabeled nucleotides and enzyme to initiate solid-phase bridge amplification.
Denature the double stranded molecules

approach.

For more details, see Mardis, 2008 or "Next-generation DNA sequencing methods."(*Annu. Rev. Genomics Hum. Genet.* 9 (2008): 387-402.), or the video: https://www.youtube.com/embed/HMyCqWhwB8E?

How Are the Read-Pairs Generated?

After generating the forward reads, specific primers are added to the outer end of the DNA fragments. These primers will attach to the primers on the cell surface, forming the bridge again. Using different restriction enzyme, we can cut another end of the “bridges”, releasing the “inner” end, i.e., the DNA fragment flips over.Then sequencing procedure repeats, resulting in **reverse reads**. As a result, we get a pair of reads – forward and reverse – which comes from the ends of the same DNA fragments, separated by a gap of known length, and oriented towards each other ("forward-reverse" orientation).



As the result, during the paired-end sequencing, each pair of reads comes from the two ends of the same DNA fragment. The total length of a read pair (computed as the sum of lengths of both reads plus the length of the gap) is referred to as the insert-size. For most applications, *InsertSize* is less than 1000 since it is difficult to generate read-pairs with larger insert sizes (they start to wobble, complicating cluster generation and detection). Nevertheless, biologists do not give up, and they invent new approaches to overcome this problem.

References

A.Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, A. V. Pyshkin, A. V. Sirotkin, N. Vyahhi, G. Tesler, M. A. Alekseyev, and P. A. Pevzner. (2012) SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*. 19: 455-477

G. M. Bennett and N.A. Moran. S. Small, smaller, smallest: the origins and evolution of ancient dual symbioses in a phloem-feeding insect. *Genome Biology and Evolution*. 5:1675-88 (2013)

A. Gurevich, V. Saveliev, N. Vyahhi and G. Tesler. (2013) QUAST: quality assessment tool for genome assemblies. *Bioinformatics* 29: 1072-1075.

E.R. Mardis (2008) Next-generation DNA sequencing methods. *Annu. Rev. Genomics Hum. Genet*. 9: 387-402.

Rohde H, Qin J, Cui Y, Li D, Loman NJ, Hentschke M, Chen W, Pu F, Peng Y, Li J, Xi F, Li S, Li Y, Zhang Z, Yang X, Zhao M, Wang P, Guan Y, Cen Z, Zhao X, Christner M, Kobbe R, Loos S, Oh J, Yang L, Danchin A, Gao GF, Song Y, Li Y, Yang H, Wang J, Xu J, Pallen MJ, Wang J, Aepfelbacher M, Yang R; E. coli O104:H4 Genome Analysis Crowd-Sourcing Consortium. (2011) Open-source genomic analysis of Shiga-toxin-producing *E. coli* O104:H4. *New England J. Medicine*, 365:718-24