Programming Assignment 3: Genome Assembly Faces Real Sequencing Data

Revision: November 15, 2018

Introduction

Welcome to the third programming assignment of the Algorithms and Data Structures Capstone! In this assignment, you will face practical challenges introduced by quirks in modern sequencing technologies and practice using algorithmic techniques that have been devised to address these challenges.

Passing Criteria: 3 out of 6

Passing this programming assignment requires passing at least 3 out of 6 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Finding a Circulation in a Network

Problem Introduction

In a circulation problem, one is given a directed graph $G(V, E)$ where each edge $e \in E$ is assigned a lower bound l_e and a capacity c_e such that $0 \leq l_e \leq c_e$. The goal is to check whether it is possible to assign a flow f_e to each edge so as to satisfy the following two conditions:

- capacity conditions: for each edge $e \in E$,

$$l_e \leq f_e \leq c_e\,;$$

- conservation of flow: for each vertex $v \in V$,

$$\sum_{(u,v)\in E} f_{(u,v)} = \sum_{(v,w)\in E} f_{(v,w)}\,.$$

The problem is similar to the maximum flow problem, but in the circulation problem we force each edge e to transfer at least l_e units of flow. Also, there is no source that produces a flow and there is no sink that absorbs a flow. For this reason, we do not maximize a flow in this problem, but simply check whether there is a flow satisfying the two types of constraints above.

Problem Description

Task. Given a network with lower bounds and capacities on edges, find a circulation if it exists.

Input Format. The first line contains integers n and m — the number of vertices and the number of edges, respectively. Each of the following m lines specifies an edge in the format "`u v l c`": the edge (u, v) has a lower bound l and a capacity c. (As usual, we assume that the vertices of the network are $\{1, 2, \ldots, n\}$.) The network does not contain self-loops, but may contain parallel edges.

Constraints. $2 \leq n \leq 40$; $1 \leq m \leq 1\,600$; $u \neq v$; $0 \leq l \leq c \leq 50$.

Output Format. If there exists a circulation, output `YES` in the first line. In each of the next m lines output the value of the flow along an edge (assuming the same order of edges as in the input). If there is no circulation, output `NO`.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	3	3	4.5	15	15	9

Memory Limit. 1024MB.

Sample 1.

Input:

```
3 2
1 2 0 3
2 3 0 3
```

Output:

```
YES
0
0
```

Sample 2.

Input:

```
3 3
1 2 1 3
2 3 2 4
3 1 1 2
```

Output:

```
YES
2
2
2
```

Sample 3.

Input:

```
3 3
1 2 1 3
2 3 2 4
1 3 1 2
```

Output:

```
NO
```

What To Do

Reduce this problem to the maximum flow problem.

2 Dataset Problem: Selecting the Optimal k-mer Size

In dataset problems, you solution is going to be tested against a *single* dataset as opposed to problems in the previous classes in this specialization. For this reason, there is no Constraints section in the problem description below. The sample section shows just a similar dataset. Your program is not going to be tested on this dataset.

Problem Introduction

As you may recall, because we are not guaranteed to be given every possible read (i.e., the substring of Genome beginning at every possible position), we cannot expect to use our reads directly and have our de Bruijn graph have an Eulerian Cycle. However, our way of combatting this is to choose some $k < \mathit{ReadLength}$ to break our reads down, and hopefully, for some value of k, the fragments will be small enough for there to exist an Eulerian Cycle in the de Bruijn graph created from the fragments, but not so small that the de Bruijn graph becomes too convoluted.

Problem Description

Task. Given a list of error-free reads, return an integer k such that, when a de Bruijn graph is created from the k-length fragments of the reads, the de Bruijn graph has a single possible Eulerian Cycle.

Dataset. The input consist of 400 reads of length 100, each on a separate line. The reads contain no sequencing errors. Note that you are **not** given the 100-mer composition of the genome (i.e., some 100-mers may be missing).

Output. A single integer k on one line.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	3	3	4.5	15	15	9

Memory Limit. 512MB.

Sample 1.

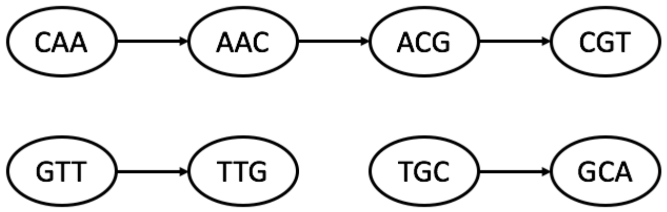
Input:

```
AACG
ACGT
CAAC
GTTG
TGCA
```

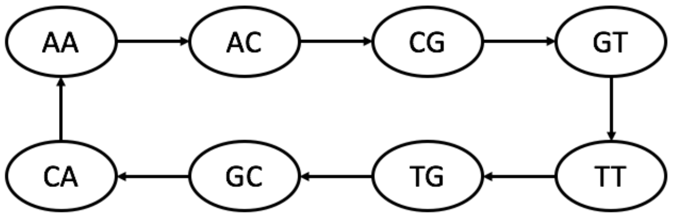
Output:

```
3
```

Below is the de Bruijn graph constructed from the input reads of length $k = 4$ (notice that there is no cycle in the de Bruijn graph because of imperfect coverage):



However, if we fragment the reads into k-mers of length $k = 3$, we'll see that the resulting de Bruijn graph does indeed have a cycle with which we would be able to reconstruct the genome:



3 Dataset Problem: Bubble Detection

In dataset problems, you solution is going to be tested against a *single* dataset as opposed to problems in the previous classes in this specialization. For this reason, there is no Constraints section in the problem description below. The sample section shows just a similar dataset. Your program is not going to be tested on this dataset.

Problem Introduction

We define a directed path in a directed graph as *short* if its length (in number of edges) does not exceed the bubble length threshold t. A path is *non-overlapping* if it traverses each of its nodes exactly once. Two paths between nodes v and w are called *disjoint* if they do not share any nodes (except for v and w). Given two distinct nodes v and w, a (v, w)-bubble is defined as a pair of short non-overlapping disjoint paths between v and w.

In this challenge, given a bubble length threshold of t, you will be given the task of detecting (v, w)-bubbles for all possible v and w in the de Bruijn graph generated from a simulated error-prone sequencing dataset.

Problem Description

Task. Given a list of error-prone reads and two integers, k and t, construct a de Bruijn graph from the k-mers created from the reads and perform the task of bubble detection on this de Bruijn graph with a path length threshold of t.

Dataset. The first line of the input contains two integers, k and t, separated by a single space. Each subsequent line of the input contains a single *read*. The reads are given to you in alphabetical order because their true order is hidden from you. Each read is 100 nucleotides long and contains a single sequencing error (i.e., one mismatch per read) in order to simulate the 1% error rate of Illumina sequencing machines. Note that you are not given the 100-mer composition of the genome (i.e., some 100-mers may be missing).

Output. A single integer (the number of (v, w)-bubbles) on one line.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	20	20	30	100	100	60

Memory Limit. 512MB.

Sample 1.

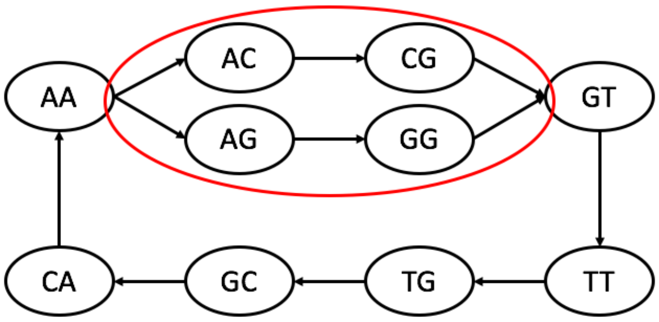
Input:

```
3 3
AACG
AAGG
ACGT
AGGT
CGTT
GCAA
GGTT
GTTG
TGCA
TTGC
```

Output:

```
1
```

Below is the de Bruijn graph constructed from the input reads after they have been broken down into 3-mers (the single (v, w)-bubble has been circled in red):



4 Dataset Problem: Tip Removal

In dataset problems, you solution is going to be tested against a *single* dataset as opposed to problems in the previous classes in this specialization. For this reason, there is no Constraints section in the problem description below. The sample section shows just a similar dataset. Your program is not going to be tested on this dataset.

Problem Introduction

Tips are 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. In this challenge, you will be given the task of removing tips from the de Bruijn graph generated from a simulated error-prone sequencing dataset.

Problem Description

Task. Given a list of error-prone reads, construct a de Bruijn graph from the 15-mers created from the reads and perform the task of tip removal on this de Bruijn graph.

Dataset. The input consist of 400 reads of length 100, each on a separate line. The reads are given to you in alphabetical order because their true order is hidden from you. Each read is 100 nucleotides long and contains a single sequencing error (i.e., one mismatch per read) in order to simulate the 1% error rate of Illumina sequencing machines. Note that you are not given the 100-mer composition of the genome (i.e., some 100-mers may be missing).

Output. A single integer (the number of edges removed during the Tip Removal process) on one line.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	20	20	30	100	100	60

Memory Limit. 512MB.

Sample 1.

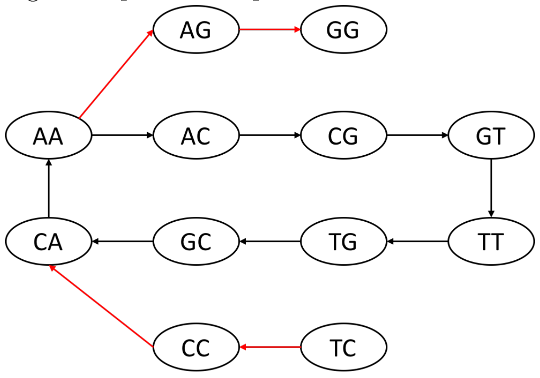
Input:

```
AACG
AAGG
ACGT
CAAC
CGTT
GCAA
GTTG
TCCA
TGCA
TTGC
```

Output:

```
4
```

Below is the de Bruijn graph constructed from the input reads after they have been broken down into 3-mers (the edges removed during the Tip Removal process have been colored red):



5 Dataset Problem: Assembling the phi X174 Genome from Error-Prone Reads using de Bruijn Graphs

In dataset problems, you solution is going to be tested against a *single* dataset as opposed to problems in the previous classes in this specialization. For this reason, there is no Constraints section in the problem description below. The sample section shows just a similar dataset. Your program is not going to be tested on this dataset.

Problem Introduction

In this challenge, you will be given the task of performing Genome Assembly on a simulated error-prone sequencing dataset using de Bruijn graphs.

Problem Description

Task. Given a list of error-prone reads, perform the task of Genome Assembly using de Bruijn graphs and return the circular genome from which they came. Break the reads into fragments of length $k = 15$ before constructing the de Bruijn graph, remove tips, and handle bubbles.

Dataset. Each line of the input contains a single *read*. The reads are given to you in alphabetical order because their true order is hidden from you. Each read is 100 nucleotides long and contains a single sequencing error (i.e., one mismatch per read) in order to simulate the 1% error rate of Illumina sequencing machines. Note that you are not given the 100-mer composition of the genome (i.e., some 100-mers may be missing).

Output. Output the assembled genome on a single line.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	10	10	15	50	50	30

Memory Limit. 512MB.

Sample 1.

Input:

```
AAC
ACG
GAA
GTT
TCG
```

Output:

```
ACGTTCGA
```

In this sample, the circular genome is ACGTTCGA (the sample output), and the reads were all generated from the genome: ACGTTCGA, ACGTTCGA, ACGTTCGA, ACGTTCGA, and ACGTTCGA. Note that we did not put mismatches in these reads because they are extremely short, so introducing the sequencing errors on such short reads would only make the solution difficult to see. On the real dataset, however, each read (of length 100) will have exactly one error.

6 Final Project: Implementing an Assembler

Problem Introduction

Now that you have solved the individual components of a genome assembler, it is time to put all of the pieces together. Using the de Bruijn graph approach to genome assembly as well as using all of the various error-correction techniques you have learned about, your task is to implement an assembler that takes as input reads or read-pairs and outputs the assembled contigs.

Note that, for read-pairs, there is typically a (roughly) known distance between the two reads: the distance is known by design of the sequencing experiment, but further explanation is out of the scope of this project. Also, note that we say "roughly" because, although biologists *attempt* to keep the distance of all read-pairs constant, there is slight deviation in distances. For example, if you were to design an experiment where you expect all read-pairs to have a distance of 400 bases between the two reads of the pair, in reality, you would observe some with distance 400, some with 401, some with 399, etc.

Also note that, in real sequencing experiments, each base of a read has a quality score associated with it, which effectively tells us how confident the sequencing machine is in calling the given base. Further explanation is out of the scope of this project, but feel free to read about the FASTQ file format and Phred-scaled quality scores. Basically, because of limitations in the sequencing technologies, reads tend to drop off in quality closer to their 3' ends. As a result, it is common practice for biologists to truncate reads once the quality drops off too far so we only attempt to assemble high quality segments of sequence data. As a result, when dealing with real datasets, the reads that are originally sequenced are all of equal length, but the truncated reads used in the assembly typically vary in length.

Of course, there are various metrics used to assess the quality of a genome assembly. One such metric is N50, the length for which the collection of all contigs of that length or longer covers at least 50% of assembly length. Note that the N50 metric does not depend on any extranneous information: it only depends on measurements of the alignment itself. A slightly better metric, which can only be used if the reference genome is known, is NG50, which is identical to N50, except instead of looking at coverage of at least 50% of the assembly length, we look at coverage of at least 50% of the reference genome. The last metric we will discuss is NGA50 (which is the metric we use to assess the quality of your assembler in this challenge), which is identical to NG50, except instead of simply looking at contigs with respect to the reference genome, we look at aligned blocks of the assembled genome against the reference genome.

In this challenge, we will be feeding your assembler multiple datasets: error-free as well as error-prone, single reads as well as read-pairs, and even a real WGS dataset. We have run each dataset through the SPAdes assembler and used QUAST to obtain NGA50 values for each assembly. We grade your assembler based on how well the NGA50 of the assembly generated by your assembler compares to that generated by SPAdes.

Problem Description

Task. Given a list of reads or read-pairs, generate all contigs in their assembly using the de Bruijn graph approach as well as all of the error-correction techniques you have learned.

Input Format. In this challenge, we generate t reads by randomly selecting k-mers from the genome. In the case of read-pairs, we randomly select the first read (just as with single reads), and then we randomly choose the desired distance from a uniform distribution centered around d, and using this distance, we choose the second read in the pair.

Some datasets may be error-free, and some may be error-prone. For the error-prone datasets, we randomly choose a single site to randomly mutate in each read.

The first line of the input is an integer t representing the number of reads or read-pairs that will follow (i.e., how many lines of input you will have to parse following the first line). Each subsequent line of the input contains a single read or a single read-pair. If the dataset constains single reads (i.e., not read-pairs), each line of sequence data will be a single k-mer. If the dataset contains read-pairs, each line of sequence data will have three parts: the first k-mer, the second k-mer, and d (the distance we *expect* to occur between the two reads, i.e., the d used as the center of the uniform distribution from which the true distance between the two reads was sampled), and these three components will be separated by the | symbol. In other words, each line will look like READ1|READ2|d (but as mentioned, note that the d listed might not be the actual distance between the two reads).

For the single-read datasets generated from N. deltocephalinicola, you will be given almost $t = 34\,000$ reads, each of length $k = 100$. For the single-read datasets generated from E. coli X, you will be given almost $t = 1\,400\,000$ reads, each of length $k = 100$. For the read-pair datasets generated from E. coli X, you will be given almost $t = 700\,000$ read-pairs, where each read is of length $k = 100$. For the real dataset N. deltocephalinicola, you will be given almost $t = 19\,679$ read-pairs, where the reads vary in length.

Output. Output all contigs in the assembly constructed from the reads (or read-pairs). Format your output in the FASTA format. The identifiers of your FASTA file do not matter.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	300	300	450	1500	1500	600

Memory Limit. 4096MB.

Sample 1.

Input:

```
8
ATG
ATG
TGT
TGG
CAT
GGA
GAT
AGA
```

Output:

```
>CONTIG1
AGA
>CONTIG2
ATG
>CONTIG3
ATG
>CONTIG4
CAT
>CONTIG5
GAT
>CONTIG6
TGGA
>CONTIG7
TGT
```

7 Solving a Programming Challenge in Five Easy Steps

7.1 Reading Problem Statement

Start by reading the problem statement that contains the description of a computational task, time and memory limits, and a few sample tests. Make sure you understand how an output matches an input in each sample case.

If time and memory limits are not specified explicitly in the problem statement, the following default values are used.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	1.5	5	5	3

Memory limit: 512 Mb.

7.2 Designing an Algorithm

After designing an algorithm, prove that it is correct and try to estimate its expected running time on the most complex inputs specified in the constraints section. If you laptop performs roughly 10^8–10^9 operations per second, and the maximum size of a dataset in the problem description is $n = 10^5$, then an algorithm with quadratic running time is unlikely to fit into the time limit (since $n^2 = 10^{10}$), while a solution with running time $O(n \log n)$ will. However, an $O(n^2)$ solution will fit if $n = 1\,000$, and if $n = 100$, even an $O(n^3)$

solutions will fit. Although polynomial algorithms remain unknown for some hard problems in this book, a solution with $O(2^n n^2)$ running time will probably fit into the time limit as long as n is smaller than 20.

7.3 Implementing an Algorithm

Start implementing your algorithm in one of the following programming languages supported by our automated grading system: `C`, `C++`, `Haskell`, `Java`, `JavaScript`, `Python2`, `Python3`, or `Scala`. For all problems, we provide starter solutions for `C++`, `Java`, and `Python3`. For other programming languages, you need to implement a solution from scratch. The grading system detects the programming language of your submission automatically, based on the extension of the submission file.

We have reference solutions in `C++`, `Java`, and `Python3` (that we don't share with you) which solve the problem correctly under the given constraints, and spend at most 1/3 of the time limit and at most 1/2 of the memory limit. You can also use other languages, and we've estimated the time limit multipliers for them. However, we have no guarantee that a correct solution for a particular problem running under the given time and memory constraints exists in any of those other languages.

In the Appendix, we list compiler versions and flags used by the grading system. We recommend using the same compiler flags when you test your solution locally. This will increase the chances that your program behaves in the same way on your machine and on the testing machine (note that a buggy program may behave differently when compiled by different compilers, or even by the same compiler with different flags).

7.4 Testing and Debugging

Submitting your implementation to the grading system without testing it first is a bad idea! Start with small datasets and make sure that your program produces correct results on all sample datasets. Then proceed to checking how long it takes to process a large dataset. To estimate the running time, it makes sense to implement your algorithm as a function like `solve(dataset)` and then implement an additional procedure `generate()` that produces a large dataset. For example, if an input to a problem is a sequence of integers of length $1 \le n \le 10^5$, then generate a sequence of length 10^5, pass it to your `solve()` function, and ensure that the program outputs the result quickly.

Check the boundary values to ensure that your program processes correctly both short sequences (e.g., with 2 elements) and long sequences (e.g., with 10^5 elements). If a sequence of integers from 0 to, let's say, 10^6 is given as an input, check how your program behaves when it is given a sequence $0, 0, \ldots, 0$ or a sequence $10^6, 10^6, \ldots, 10^6$. Afterwards, check it also on randomly generated data. Check degenerate cases like an empty set, three points on a single line, a tree which consists of a single path of nodes, etc.

After it appears that your program works on all these tests, proceed to stress testing. Implement a slow, but simple and correct algorithm and check that two programs produce the same result (note however that this is not applicable to problems where the output is not unique). Generate random test cases as well as biased tests cases such as those with only small numbers or a small range of large numbers, strings containing a single letter "a" or only two different letters (as opposed to strings composed of all possible Latin letters), and so on. Think about other possible tests which could be peculiar in some sense. For example, if you are generating graphs, try generating trees, disconnected graphs, complete graphs, bipartite graphs, etc. If you generate trees, try generating paths, binary trees, stars, etc. If you are generating integers, try generating both prime and composite numbers.

7.5 Submitting to the Grading System

When you are done with testing, submit your program to the grading system! Go to the submission page, create a new submission, and upload a file with your program (make sure to upload a source file rather than an executable). The grading system then compiles your program and runs it on a set of carefully constructed tests to check that it outputs a correct result for all tests and that it fits into the time and memory limits. The grading usually takes less than a minute, but in rare cases, when the servers are overloaded, it might take longer. Please be patient. You can safely leave the page when your solution is uploaded.

As a result, you get a feedback message from the grading system. You want to see the "`Good job!`" message indicating that your program passed all the tests. The messages "`Wrong answer`", "`Time limit exceeded`", "`Memory limit exceeded`" notify you that your program failed due to one of these reasons. If you program fails on one of the first two test cases, the grader will report this to you and will show you the test case and the output of your program. This is done to help you to get the input/output format right. In all other cases, the grader will *not* show you the test case where your program fails.

8 Appendix: Compiler Flags

C (`gcc 5.2.1`). File extensions: `.c`. Flags:

```
gcc -pipe -O2 -std=c11 <filename> -lm
```

C++ (`g++ 5.2.1`). File extensions: `.cc`, `.cpp`. Flags:

```
g++ -pipe -O2 -std=c++14 <filename> -lm
```

If your `C/C++` compiler does not recognize `-std=c++14` flag, try replacing it with `-std=c++0x` flag or compiling without this flag at all (all starter solutions can be compiled without it). On Linux and MacOS, you most probably have the required compiler. On Windows, you may use your favorite compiler or install, e.g., `cygwin`.

Java (`Open JDK 8`). File extensions: `.java`. Flags:

```
javac -encoding UTF-8
java -Xmx1024m
```

JavaScript (`Node v6.3.0`). File extensions: `.js`. Flags:

```
nodejs
```

Python 2 (`CPython 2.7`). File extensions: `.py2` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python2"). No flags:

```
python2
```

Python 3 (`CPython 3.4`). File extensions: `.py3` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python3"). No flags:

```
python3
```

Scala (`Scala 2.11.6`). File extensions: `.scala`.

```
scalac
```