CS466 Programming Assignment

1. Overview

This assignment asks you to implement either (your choice) of two algorithms learned in this class: 1) the core of the BLAST algorithm, or 2) de Bruijn graph assembly. You can choose to code in any language that you are comfortable with, but your submitted files should be self-contained (with the exception of third-party libraries, see below). The grading will largely be based on your implementation having the correct idea and outputting meaningful results instead of focusing on accuracy or running-time. With that said, we plan to use an autograder to simplify the grading process, but the results of the autograder will only serve as a shortcut for us to determine the quality of your implementation. Your final score may very well disagree with the autograder if we think that good effort has been put into the implementation and the results are meaningful.

1.1. Directory Structure

While the autograding platform and concrete submission directions are still being worked on, you can prepare your submission directory to have a main entry file named either blast (with the appropriate capitalization and extension determined by your language) or assembly (again, with the appropriate capitalization and extension) depending on the option you choose. Here are some examples of how things will be run for some common languages:

For the BLAST option:

```bash
# assuming the current working directory is inside your source directory
python blast.py <queries> <db> # Python 2.7
python blast.py <queries> <db> # Python 3.8
javac *.java && java Blast <queries> <db> # Java 9
```

g++ -g -O2 -std=gnu++17 *.cpp -o blast && ./blast <queries> <db> # C++

For the de Bruijn graph assembly option:

```bash
# assuming the current working directory is inside your source directory
python assembly.py <reads> # Python 2.7
python assembly.py <reads> # Python 3.8
javac *.java && java Assembly <reads> # Java 9
```

g++ -g -O2 -std=gnu++17 *.cpp -o assembly && ./assembly <reads> # C++

The exact format and input structure will be outlined in the following sections. If you use other languages, you can expect analogous commands used.

1.2. Third-Party Libraries and Build Tools

Third-party libraries can be used for data-processing and data structures, but the core of the algorithm (for example, finding the Eulerian path) should be entirely implemented by you. If you use third-party libraries, you are encouraged to use some kind of cross-platform build tool (say, CMake for C++, Gradle for Java, Poetry for Python), but this will not be an requirement. In general, just be prepared to document
how to compile and run your code during submission (conforming to some convention makes the autograding easier, but it will not be enforced strictly).

For Python, the latest stable version of networkx can be assumed to be pre-installed during autograding (this is not an endorsement for networkx. It is not necessary, but some people might be familiar with it and want to use it).

1.3. Starter Code

Here is some very minimal starter code for Python3 that will handle reading the inputs for you: https://git.io/JOGAa. Feel free to use it (and feel free to not use it.)

2. BLAST

The goal here is to implement the core of the BLAST algorithm (using neighborhood words, extending the seed alignments, etc.) to find good local alignments of query sequences against a single database sequence.

The submitted program should receive two filenames from the command line arguments (ARGV). The first filename points to a file containing the query sequences. This query file contains one sequence per line, and they denote the BLAST queries. The second filename points to the database file. This database will only contain one sequence (the subject sequence), and it will be on the first line in the database file. All the sequences here are nucleotide sequences with alphabets ACTG.

Here are some assumptions that will be useful: the subject sequence will always be at least twice as long as the longest query sequence, and its length will never exceed 10kb. The query sequences will have lengths between 30 and 55. Be prepared for inputs containing ~500 query sequences.

You are free to select your own parameters for BLAST as long as they make sense (we do not score by accuracy), but if you are looking for something to aim for, having most of the queries where the query sequence is at most two edits (insertion, deletion, mutation) away from some substring of the subject sequence properly aligned can be a good thing to try.

For the output, the program should for each output local alignment, print (to standard output) a comma separated line containing the following values (all indices outputted here are 1-based): the index of the query sequence this search result corresponds to, the index where the local alignment starts in the query sequence, the index where the local alignment ends in the query sequence, the index where the local alignment starts in the subject sequence, and the index where the local alignment ends in the subject sequence.

For example, suppose that this is the input given to the program (ran by <your_program> queries.txt db.txt):

queries.txt

TTTGAAGTGTACTCTCGTCTACTTAAGGC

db.txt

CGGCCACCAGCAATGATCGCAGTCGTTAAAAGTGTTACTCTCGTCTACTTAAGGCAGGTTACGCAGCGCCGCTG
The intended output is given below (which means that the first sequence is aligned (from pos 1 to 31) to the subject sequence (from 26 to 56):

**output**

1,1,31,26,56

This corresponds to the following alignment obtained by blastn:

```
Query   1   TTTGAAGGTACTCTCCGTACTTAAGGC   31
TTTGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
Sbjct   26  TTTGAAGGTACTCTCCGTACTTAAGGC   56
```

Here is another set of input/output:

**queries.txt**

```
TTTGAAGGTGTTTCCTCCGTACTTAAGGC
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
TTTGAAGGTACTCTCCGTACTTAAGGC
```

**db.txt**

```
GGCCACCCGGAATGATCGATCGTTTTGAAGGTGTTACTCTCCGTACTTAAGGCAGGTACGGCGCCGCGCTG
```

**output**

```
1,1,31,26,56
3,1,38,26,56
```

*We also provide a set of input that strongly resembles what the autograder will use: see it [here](#)*

Note that depending on the parameters chosen, it is entirely possible for a correct implementation to have outputs different from these example inputs/outputs.

### 3. De Bruijn Graph Assembly

The goal here is to implement the de Bruijn Graph assembly algorithm for genome assembly.

The program that is submitted should receive one filename from the command line arguments (ARGV). The filename should point to a file containing the reads to be assembled separated by new lines. All the sequences here are assumed to be nucleotide sequences with alphabets ACTG. Each sequence will always be of the same length $k$. This implies that your de Bruijn graph should have $(k - 1)$mers as the nodes.

The output is simply the assembled genome from the input reads (if there are multiple equally valid assembled genomes, output any of them) using the reads, outputted (to standard output) in one line. If you cannot assemble the genome using the input reads, output a single line `-1` instead.

Be prepared to assemble up to 2000 reads with length no more than 60bp. The final assembled genome length will not exceed 10kb. For the inputs that can be assembled, the reads will be entirely error free (i.e. each read will be a substring of the assembled
genome), and every kmer (the $k$ will be fixed throughout one set of read, i.e. throughout one execution of your program, but may vary between different input files) at each position will appear in the reads exactly once (if a kmer appears at exactly two different positions in the genome, then it will be included exactly twice in the input reads).

Here are some example inputs and outputs (program invoked by `<your_program> reads.txt`):

reads.txt
AAA
AAC
ACC
CCC
CCA

output
AAACCCA

Here is another set of input and output:

reads.txt
TGACTG
GACTGG
ACTGGA
CTGGAT
TGGATC
GGATCG
AAACCA
AACCAC
ACCACT
CCACTG
CACTGA
ACTGAC
CTGACT
GATCGA
ATCGAT
TCGATC
CGATCG

output
AAACCACTGACTGGATCGATCG

A set of input/output demonstrating what to do when assembly is not possible:

reads.txt
AAAAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCCCCCC

output
-1
We also provide a set of inputs that strongly resembles what the autograder will use:
see it here.

1: Perhaps just give us a heads up if you plan to use anything that is more obscure than Haskell or Julia. 😄