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[1], 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 (update: the submission directions has been posted to Piazza. Also see the new section 1.4 for more information), 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:

```
# assuming the current working directory is inside your source directory
python blast.py <queries> <db> # Python 2.7
python3 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:

```
# assuming the current working directory is inside your source directory
python assembly.py <reads> # Python 2.7
python3 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, bridge detection, etc.) 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.)

1.4. Grading Information

See also the original post on Piazza on submission instructions. Feel free to ask me questions or drop suggestions on the autograder or the grading of this assignment.

1.4.1. Grading standards

In the case that the submitted code conforms to the autograder convention, the autograder's score serves as a lower bound for the final score (unless the code is really abusing this somehow by special casing the autograder inputs). After serving as this lower bound, the autograder score is no longer directly considered (the failed test cases might help pinpointing mistakes).

Afterwards the grading is based on reading the code (and blackbox testing the code), for each major programming mistake that reflects some misunderstanding of core parts of the algorithms, 10 pts is deducted. For other types of mistakes not directly related to the understanding of the algorithm (say due to accidentally introducing UB in your C++ code), at most 5 pts is deducted.

In the special case that the code is not finishing on the autograder input simply due to an inefficient implementation (since the autograder should have given plenty of time for an $O(n^2)$ algorithm to finish), I will try running the code (on my laptop) with a total 25 minutes allocated on all inputs used by the autograder, if the code finishes within 25 minutes, no points will be deducted. Otherwise I am going to check if the submitted code on whether it is using this much time due to an inefficient (probably $\omega(n^3)$) but correct algorithm (in which case -7 pts), or it is not implementing the algorithm right, in which case the previous paragraph applies.

Only in the case that the submitted code does not conform to the autograder convention that you need to document how to run your code and the format your code uses. In all cases, the code itself can be unexplained/uncommented.

1.4.2. Constantly not passing some of the test cases

This applies more for the de Bruijn assembly option. It might help to first convince yourself that the algorithm is correct, but you might not be handling some cases correctly (for example for the assembly option, the existence of a Eulerian cycle as a subgraph of the entire de Bruijn graph). Make a Piazza post about it and maybe I can help.

1.4.3. Timing out on the autograder

The autograder should have allowed plenty of time (1200s) for a simple algorithm to finish. See 1.4.1 for more information on how the grading

will be handled in this case. For now (before I really have time to implement time-outs in the autograder), it is necessary to run the tests yourself to pinpoint where the code is taking too much time. Not having time-outs for individual test cases is not ideal for an autograder and is something I should work on if I were to be doing this again.

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

```
TTTGAAGTGTTACTCTCCGTCTACTTAAGGC
```

db.txt

```
CGGCCACCGGCAATGATCGCAGTCGTTTGAAGTGTTACTCTCCGTCTACTTAAGGCGAGGTACGCGCCGCTG
```

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   TTTGAAGTGTTACTCTCCGTCTACTTAAGGC  31
           ||||||||||||||||||||||||||||||||
Sbjct  26  TTTGAAGTGTTACTCTCCGTCTACTTAAGGC  56
```

Here is another set of input/output:

queries.txt

```
TTTGAAGTGTTTCTCTCCGTCTACTTAAGGC
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
TTTGAAGTGTTACTCTCCGTCTACTTAGGC
```

db.txt

```
CGGCCACCGGCAATGATCGCAGTCGTTTGAAGTGTTACTCTCCGTCTACTTAAGGCGAGGTACGCGCCGCTG
```

output

```
1,1,31,26,56
3,1,30,26,56
```

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 (edit: in retrospect I think I generated test cases that exceeded this bound of # of reads by a factor of 5. Unfortunately it seems that this is hard to compensate for at this point, I upped the Gradescope timeout limit and compute power to compensate for this. If your code is just slow on those large inputs no penalty will be applied since the input sizes are large now (~10k reads now, still permitting $O(n^2)$ algorithms to finish well within limit)). 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

```
AAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCCCC
```

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. ↩