PR3: Genome Assembly

CS 182/282 Spring 2022

Released: Thursday, March 14th, 2024

Due: 11:59pm on Thursday, April 4, 2024

Overview

In the previous PR, you developed methods for quality control of sequencing data; namely, the implementation of algorithms to remove contamination from reads and correct errors due to missequencing of bases. In this PR, you will implement your own genome assembler which can take in a set of reads from a text file and output a de Bruijn graph representation of the full alignment and infer the most likely target sequence associated with those reads. Note that the assembler implemented in these PRs is a simplified version of an actual genome assembler, but involves many of the important concepts underlying genome assembly.

Genome assembly (and bioinformatics in general) continues to be at the forefront of disease research. It can often be an important starting point for further analysis into an organism's genome, or even a virus's. You'll have the chance to experiment with assembly on real genomes in the application section of this project!

Reading

- How to apply de Bruijn graphs to genome assembly (Compeau et al., 2011)

Gradescope

To hand in your project, submit all of your project files to Gradescope. Make sure that any required shell scripts are in the root directory of your submission (or uploaded as individual files). We have made a small number of test cases visible to you after you submit so that you can ensure your program is working properly.

Our Gradescope autograder is currently configured to accept solutions written in Python 2, Python 3 (packages: `numpy`, `pandas`, `networkx`), Java, and R. If you plan to use a different programming language that is installed on the department machines, you must let the TA staff know as soon as possible so we can update our autograder settings.

Extraneous print statements (i.e. anything besides the specified terminal outputs) will break the autograder. Make sure to remove such code before submitting.

P1: de Bruijn Graph Inference (50 points)

Your first task will be to implement a de Bruijn graph sequence assembler.

Assumptions

For this problem, you may make the following assumptions about the input data:

- Reads will cover the entire target sequence (there will be no missing information)
- Reads will not result in de Bruijn graphs which contain cycles
- All reads will be k-mers of the same constant length k
- There will be no edges which lead into a possible starting node, or out of a possible ending node, of the final de Bruijn graph
- The length of a putative target sequence is proportional to its likelihood of correctness

Implementation

You should submit a shell script called `debruijn.sh` which can be run as follows:

```
$ sh debruijn.sh <reads.txt>
```

where `reads.txt` is a text file containing a list of DNA reads of constant length (separated by newlines).

When called as above, the script should read in the k-mers, convert them into the edges of an internal representation of a de Bruijn graph, simplify this internal representation, and finally print out all possible most likely (i.e., maximum-length) target sequences which can be inferred from the input reads (ordered alphabetically, one per line). Your program should also generate a DOT file called `debruijn.dot` representing the **simplified** de Bruijn graph. **It is only required that you simplify singletons** (forks and crosses can remain in your graph).

Graph Visualization

DOT is a graph description language which is useful for generating automatic visualizations of graphs. DOT files are simply text files which describe nodes and edges of a graph. Your `debruijn.dot` file should contain the internal representation of your simplified de Bruijn graph. We recommend following these steps to create your DOT file:

- To initialize a graphviz digraph, use `graph = graphviz.Digraph("deBruijn")`
- Add edges to your graph using `graph.edge(node_name_1, node_name_2, edge_label)`. Note that you do not have to initialize the nodes separately. Calling `graph.edge()` does this for you.
- Write the contents of `graph.source()` to a file named `debruijn.dot` from within you program just as you would to a text file.

To visualize your DOT graph, run the command `dot -Tpng debruijn.dot -o debruijn.png`. This will export a visualization of the DOT graph as a PNG in a new file called `debruijn.png`.

If you are not using a department machine, you may need to download graphviz in order to use DOT.

Test Cases

The following test cases are intended to give you a general idea of how your program should function. You are **strongly encouraged** to develop your own test cases for this problem. Small example sequences are relatively straightforward to break down into reads and provide an easy way to test your program on a variety of inputs!

For checking your DOT outputs, you should ensure that your internal graph representations are equivalent to those shown on the test cases below. The string contents of the file need not match exactly, but the visualized graph should.

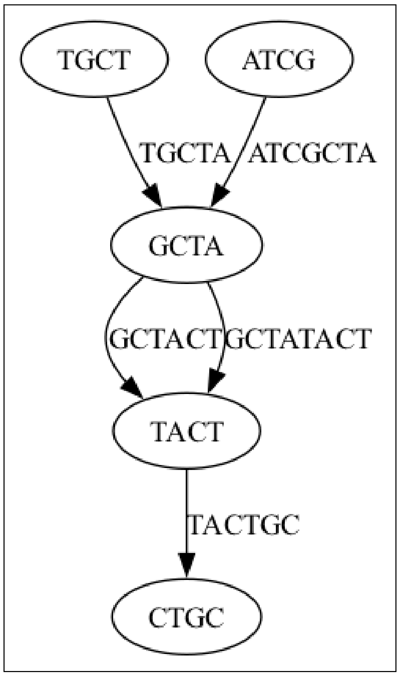
Test Case 1

```
$ sh debruijn.sh reads1.txt
ATCGCTATACTGC
```

`debruijn.dot` (tabs added for ease of reading)

```
digraph deBruijn {
        TGCT -> GCTA [label=TGCTA]
        ATCG -> GCTA [label=ATCGCTA]
        GCTA -> TACT [label=GCTACT]
        TACT -> CTGC [label=TACTGC]
}
```

`debruijn.png` (after exporting DOT → PNG)



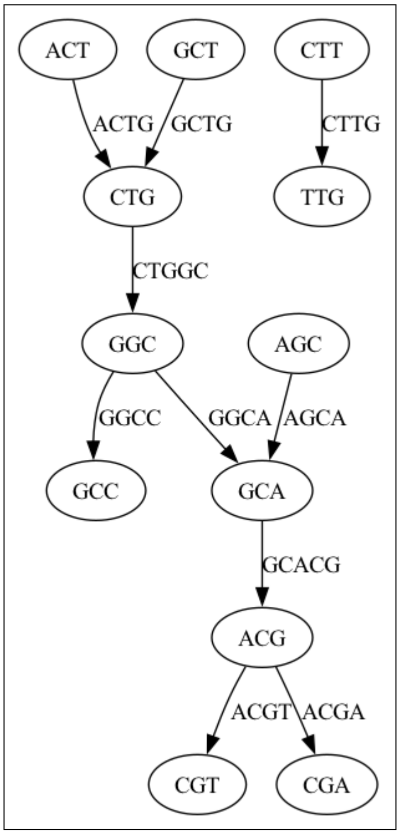
Test Case 2

```
$ sh debruijn.sh reads2.txt
ACTGGCACGA
ACTGGCACGT
GCTGGCACGA
GCTGGCACGT
```

`debruijn.dot` (tabs added for ease of reading)

```
digraph deBruijn {
        ACT -> CTG [label=ACTG]
        GCT -> CTG [label=GCTG]
        CTT -> TTG [label=CTTG]
        ACG -> CGT [label=ACGT]
        AGC -> GCA [label=AGCA]
        ACG -> CGA [label=ACGA]
        GGC -> GCA [label=GGCA]
        GGC -> GCC [label=GGCC]
        CTG -> GGC [label=CTGGC]
        GCA -> ACG [label=GCACG]
}
```

`debruijn.png` (after exporting DOT → PNG)



P2: Assembly (50 points)

Your second task will be to synthesize your prior algorithms to build a versatile genome assembler which can decontaminate reads, correct sequencing errors, construct a de Bruijn graph, and finally infer the original sequence. You should submit a shell script called `assembly.sh` which can be called as follows:

```
$ sh assembly.sh <reads.txt> <vector.txt>
```

This problem is designed to adapt your previous programs into a unified workflow. When called, the script should print out its most likely inference as described in P1 above. You do not need to generate a DOT file for this problem.

Problem

`sampleReads.txt` contains a list of reads of uniform length from the sequencing of the SARS-CoV-2 genome. These reads contain both randomly-distributed sequencing errors and contamination from a known source (`vector.txt`). Your task is to decontaminate/correct these reads and infer the sequence from which they originated. You will receive 30 points (autograded) for recovering the exact target sequence.

Some notes on integration of prior PRs are as follows:

- BLAST: a BLAST-like approach may be appropriate to scan for contamination at the ends of the reads. You may use any scoring scheme implemented previously, or devise your own.
- Contamination: when contamination is detected, simply discard the entire read. If performed correctly, this should not compromise the integrity of the assembly. Approximately 10% of the reads contain contamination.
- Correction: sequencing errors may occur at any position in any read across the entire dataset with a 0.01% error rate. Just as you did for PR2, think about how you can optimize the values of k, t, and d for this correction task.

You can set up your shell script however you'd like as long as it only takes in the reads and vector .txt files (you can call several scripts sequentially within your shell script or just one). It may help to treat this problem as a sequential process which employs different modules or functions from your other programs as appropriate.

Please note that you will likely have to modify your assembly code to make it effective at assembling simulated sequencing reads.

Solution

You should submit a file called `assembly.pdf` in which you briefly describe each of your parameter choices and complete the following Waterman Statistics questions.

Parameters

The following parameter choices are required:

- **k-mer size for contamination removal** - This represented the minimum length of contamination to be removed in PR2. While you will discard the entirety of a read if you detect contamination, this parameter is still important for accurately detecting contamination in the first place.
- **k-mer size for error correction** - This is the k-value that you use to classify k-mers as frequent or infrequent.
- **t-value for correction** - This is the threshold for classifying k-mers as frequent/infrequent
- **d-value for correction** - This is the number of differences between a frequent and an infrequent k-mer that is permitted when choosing frequent k-mer replacements for infrequent k-mers.

If you add any additional tune-able parameters to your program, please describe what they are and provide an explanation for your choice of this parameter as well.

When describing each parameter choice, please consider at least one of the following factors.

- **How does the quality of your results change based on your parameter choice?** Please provide quantitative evidence, for instance number of contaminated reads removed.
- **Why is your parameter choice statistically appropriate for this assembly task?** For example, think about the likelihood of uniqueness of the different k-mer types involved in the whole assembly process.
- **Does you parameter choice improve runtime?** This does not have to be a quantitative analysis, but if you observe that your assembly runs faster with this parameter choice, explain why you think that is.

The above factors are just suggestions of things to consider. You can absolutely justify your choice using your own line of reasoning.

Waterman Statistics

In this section of your application report, we want you to dive into the approximation of mean contig length we saw in class using the HIV genome (mutated) reads we give you in the project support files. In particular we would like to see a thoughtful discussion that cites results from your assembler on the following questions:

- **What are the challenges to computing mean contig length from your assembler when the input reads have sequencing errors?** Please provide both an answer to this question and a proposal for computing mean contig lengths in the questions to follow in a few sentences.
- **Using maximum contig length as a "biased-up" proxy for mean contig length, how does mean contig length vary with coverage?** To answer this question, you'll need to think about how to vary your coverage, and give results for $c = 1, 2, 3, 4$ (show us the math you used to make this happen!). Feel free to set a seed for replication purposes, but you'll need to use some randomization here!

  For this question, make sure that you break each read into 25-mers (or some other *fixed* value) *after* contamination and correction and use the results to form your de Bruijn graph. You will play around with this value in the next question, but for the purposes of this question, pick a singular k and vary coverage instead. If you are running into errors at 25 with recursion depth, make sure your de Bruijn graph inference can properly handle potential cycles.

  We're cheating a little bit by using maximum contig length as a proxy for mean contig length (what if the distribution of lengths changes shape with coverage?), but no need to discuss this in your writeup. It should serve as a reasonable biased proxy for our purposes.
- **How does mean contig length vary with k, the k-mer size you use to break down your reads (as described above)?** To answer this question, you'll need to fix a coverage, and instead vary this k parameter. Vary it in whatever way you see fit! Make sure to discuss why you might find what you found, from a theoretical perspective (hint: think about how uncorrected sequencing errors might play a role). Do you trust the quality of the contigs more or less for high k?
- **With your answer to the second question in mind, evaluate the robustness of our answer to question 3 of Waterman Statistics. Where does it fail for the HIV genome? Is it ever a good approximation?** To answer this question, you'll need to revisit your varying results in question 2 (and maybe try out some more extreme values). Make sure to explain *why* the approximation fails, with mention to how we derived the approximation itself (hint: think about where in the derivation we made an approximation!).