UC Berkeley, CS 176: Algorithms for Computational Biology (Fall 2020)

RNA-Sequencing Read-Alignment Project

Instructor: Nir Yosef

Introduction

You may have heard the saying that every cell in your body has the same genome - as in, more or less, your neurons, your epithelial cells, and more all come from the same "germline" DNA. It may seem perplexing how the cells in your body can fulfill such an extraordinarily large spectrum of functions: from fending off an infection to encoding memories and processing our environment. The fundamental mechanism by which cells obtain such specific functionality is through the "expression" of only certain parts of their genome. To get an idea of which genes are being expressed by a cell, it is often enough to observe the RNA of a cell. Conveniently, there are several assays for purifying and sequencing the RNA content of bulk tissues and single cells. Still, it is an ongoing and critical computational challenge to efficiently identify which gene a given read corresponds to (in short, this is a one-to-many alignment problem).

Your task

In this project, you will implement a simple scheme for the alignment of short reads from RNA-seq to a reference transcriptome and genome. As noted above, This procedure is a critical first step in any computational analysis of RNA-seq pipeline, and clever algorithms have been designed for this purpose like Tophat2 [1], STAR [2], and Kallisto [3].

You will first implement some basic BWT functions that we covered in class, and then implement a simplified version of an aligner for RNA sequencing reads. Generally, we expect you to use algorithms covered in class though you may choose another option, as long as you describe it and that it is based on BWT.

Grading will be based on the quality of written reports and the correctness of the code. Bonus points will be awarded on speed (more on that, below). This assignment is due on XXX.

General Guidelines

- **Your submission must keep all required (skeleton) functions / classes / methods in** *project.py*. Put all the python files in *project.zip*; you can create and import other files for helper functions. Do not put your util files in subdirectories, just put them in the same directory structure as *project.py*.
- You may only use Python 3. You can only use libraries and modules that are provided by the default installation of Python, as well as numpy.
- Do not change the provided function signatures and return types. We will feed you the inputs and evaluate your outputs following the defined signature and return types.
- We suggest using *jupyter notebook* for loading in the data and developing / evaluating your alignment algorithm; it's nice for keeping your variables saved if your code crashes somewhere. Once you have a working solution then port it back to the Python file.

Part 1: Implement BWT

In the first part of the project, you will be implementing functionality to support the BWT according to lectures 4-8. The skeleton code is stored in *project.py*. You will have to complete the following functions:

- `get_suffix_array`
- `get_bwt`
- `get_F`
- `get_L`
- `get_M`
- `get_OCC`
- `exact_suffix_matches`

Generally, you'll be performing the BWT for a string S by first constructing a suffix array of the string S. However, we do **not** expect you implement the KS algorithm; instead, you can rely on the fact that in our example genome the majority of suffix pairs (i, j) have an $LCP(i, j) < 100$.

Rules and Tips for Part 1

- We expect 0-indexed solutions, and any intervals are [inclusive, exclusive).
- You will solely be graded on correctness on small or medium sized examples (probably length < 10000). Your function should not time out unless it *reallllly* takes too long to run.
- Assume an alphabet consisting of ['$', 'A', 'C', 'T', 'G'], where "$" will always only be the terminator.
- Assume the s parameter in all the function is already terminated by "$".
- For constructing the suffix array, we do not suggest implementing the KS algorithm unless you want to for fun and implement it so it have good enough performance. **Constructing the suffix array should be efficient** if you want to use it for the aligner part of the project (see performance requirement details about the aligner). Also, when you use your function for the aligner, it should not use a radix of over 100 for sorting, as we will be limiting your memory usage to something reasonable, not to mention that longer radixes take longer to generate as well.
- **Do not delete docstring for** *exact_suffix_matches* if you want a sanity check of your code (see Testing below).

Testing

We have provided a simple python doctest as sanity check for your BWT functions. You can run this by running "python -m doctest project.py" on the command line.

Grading

Your code will be graded on the correctness of the BWT implementation. We also require that the runtime of this code is sane, which is taken into account for the grading of Part 2.

Part 2: Alignment of short reads from RNA sequencing

You are given a genome reference of roughly 11 million bases, with a location of all genes in the genome (each with its isoforms and exons which are expressed in the transcriptome). In addition, you are given a set of DNA sequences, each of length 50, which represent the results of RNA sequencing from that genome. You goal is to map those reads back to their place of origin in the genome. You can use Tophat2 [1], STAR [2] or any other methods of your choice or your own design that uses (at least to some extent) the BWT

Things to Note for Part 2

- You may assume that the genome only exist on one strand and that the reads come from that strand (i.e. no need to test for reverse complements).
- There are no insertions or deletions (but there could be mismatches) in the reads (i.e. each position in the read corresponds to some position in the genome).
- The annotation of genes you are provided with is incomplete. Namely, the genes / isoforms / exons that some reads are generated from have been hidden.
- Some reads represent experimental artifacts. These were randomly generated by your course' staff and do not have a good match in the genome.
- In any **true** read generated from the transcriptome (visible or unknown), there are at most **6** mismatches, so any alignment with more than 6 mismatches is invalid. Therefore, in an ideal solution, the junk reads should be reported as invalid. *See Evaluation / Scoring for more details on what we will evaluate you on.*
- *If you implement STAR, you can assume minimum and maximum intron sizes to be 20 and 10000. Some people use window sizes of 64000 bases from the anchor for their implementation, but you can use whatever works best.*

Codebase

Aligner **Class**

Implement the *Aligner* class in *project.py*. We will initialize your *Aligner* with genome information, call *Aligner.align* on a number of sequences to get your outputs, then evaluate the outputs.

In *Aligner.__init__*, you are given the genome sequence as well as *genes*, which is a list of *Gene* container objects that we have defined in *shared.py*. Each *Gene* contains a list of *Isoform* objects that correspond to the gene, and each *Isoform* contains a list of *Exon* objects. You should look at how these classes are defined in the file, but you cannot modify these classes.

Format of an Alignment

Since we specified that there is no insertion or deletion, an alignment of a read to the genome can be thought as k (usually k will be 1 or 2) separate ungapped alignments between the read and the genome (with some start index in the read and start index in the genome). Thus we expect you to specify your alignment as a python list of k tuples of ($\langle$read start index$\rangle$, $\langle$genome start index$\rangle$, $\langle$length$\rangle$). For example, an alignment of [(0, 2, 4), (3, 26, 14)] specifies that the 0th position of the read aligns to the 2nd position of the genome for 4 bases, and then the 3rd position of the read aligns to the 26th positon of the genome for 14 bases. This means there is a gap of 20 between the two exons. If you can't find an alignment for a read, you may return an empty list [].

Warning: if the ranges of two consecutive alignment pieces overlap in the read, i.e. if $\langle$read start 1$\rangle + \langle$length 1$\rangle > \langle$read start 2$\rangle$, we will discard the second piece of your alignment and score you accordingly. We check for this case with the provided functions in *evaluation.py* (see Evaluation / Scoring).

Test Files

You are given files containing examples of what kind of genome and read sequences we might test you on. You don't technically have to do anything with this, but we **highly** recommend that you parse the files and try your algorithm on these examples since they will be representative of our evaluation set.

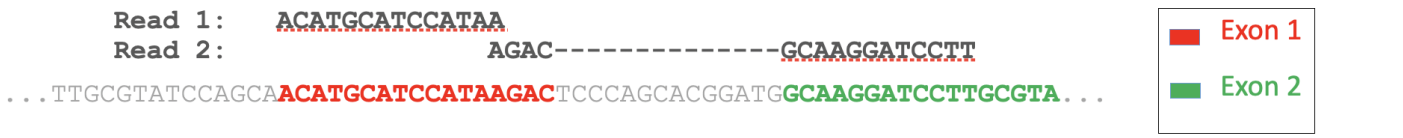
- *genome.fa* is a FASTA file with the genome sequence.
- *reads.fa* is a FASTA files with read sequences. Note that the file does not include base quality (PHRED) scores, as we have seen in the Bowtie1 algorithm. If you would like to implement Bowtie1, you can assume that the PHRED score is fixed for all bases.
- *genes.tab* is a tab-separated file containing three types of rows: a gene row begins with "gene" and specifies the *gene_id* then a semicolon-separated list of *isoform_id*; an isoform row begins with "isoform" and specifies the *isoform_id* then a sorted semicolon-separated list of *exon_id*; and an exon row begins with "exon" and specifies the *exon_id*, *start*, and *end* of the exon. Moreover, the genomic elements that are "unknown" (hidden when we test your *Aligner*) are prefixed with "unknown_". If a gene is unknown, its corresponding isoforms and exons will also be marked as unknown. **You should parse this file to construct your python set of genes that you feed into your** *Aligner.__init__*. Make sure to construct each *Gene* with all of its corresponding *Isoform* objects, and etc.

Evaluation / Scoring

Alignment Prioritization

Your should **always** prioritize aligning reads to known isoforms (in *genes.tab*) with 6 or less mismatches. Only if you can't align with 6 or less mismatches, you should try to align reads to other parts of the genome with 6 or less mismatches (as these regions may represent unknown genes). You should also minimize the number of mismatches, but do not choose an alignment to an unknown isoform with less mismatches over an alignment to a known isoform with more mismatches (but still 6 or less).

Priority 1: align read (with up to 6 mismatches) to known exons or known exon junctions



Priority 2: align read (with up to 6 mismatches) to unknown exons or unknown exon junctions

Priority 3: No alignment with up to 6 mismatches (I.e. Junk read)

You must align all indices of a read to the transcriptome. Since there is no insertion or deletion, we will be scoring your alignment based on one-to-one and **consecutive** correspondence to the transcriptome. i.e. if you match a read to an isoform, make sure that the read aligns **consecutively** to the transcript generated by concatenating the exons within the isoform, there should be no gaps when aligning to the transcriptome.

evaluation.py

To make sure you are able to evaluate your aligner's alignments, we have provided the function that we will use to evaluate your alignments in *evaluation.py*. You first need to construct a python set of known *Isoform* and a python set of unknown *Isoform* objects. You can then build an index by calling the *index_isoform_locations* function, and use this index to run *evaluate_alignment* on the alignment that you generate with Aligner.align.

Submission

This assignment is due on **Friday 11/13 11:59pm PST**. Please create an Instructional account for CS 176 and submit your project there. Only one person per pair needs to submit, but you could both submit as well just to be safe. Indicate the name of your partner when prompted by Glookup. As stated above, **your submission must keep all required (skeleton) functions / classes / methods in** *project.py*. Put all the python files in *project.zip*; you can create and import other files for helper functions. Do not put your util files in subdirectories, just put them in the same directory structure as *project.py*. You may, depending on the reason, have another chance to submit with a penalty if your first submission crashes.

Grading

You will be graded on correctness of your implementation and speed:

- 30% of your grade will be based on the correctness of Part 1.
- 50% of your grade will be based on the correctness of Part 2. 75% of genes are known, 20% are hidden and 5% are fake.
- 20% of your grade will be speed. For a 11 million base genome, your *Aligner.__init__* method must take less than 500 seconds to run. Your *Aligner.align* method must take on average less than 0.5 seconds per read. In addition, you will be penalized if your *Aligner.align* is more than 2 times slower than the mean runtime of everyone in the class.
- Bonus (10%) will be awarded to the top 10 performers as measured by speed of the alignment.

References

[1] Daehwan Kim, Geo Pertea, Cole Trapnell, Harold Pimentel, Ryan Kelley, and Steven L. Salzberg. Tophat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, 14(4):R36, Apr 2013.

[2] Alexander Dobin, Carrie A. Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R. Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 10 2012.

[3] Nicolas L. Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature Biotechnology*, 34(5):525–527, May 2016.