DeDuper Outline

Adrian Bubie

10/11/17

DeDuper Program Outline

Goal: Given a SAM file of uniquely mapped reads, remove all PCR duplicates sunc that there is only a single instance of each read.

Problem Description:

PCR duplicates are two or more reads that come from the same molecule of DNA. We want to remove PCR duplicates because they can lead to an overrepresentation of one particular gene fragment among all our reads.

Duplicates can be difficult to spot: though duplicate reads will share the same chromosome, position and strand (if non-specific) information in a SAM file following read alignment, the presence of soft-clipping and use of UMIs make it more difficult than just comparing all reads against each other and removing matches.

Additional complexity comes with paired-end reads and strand specificity, but the current scope of the problem does not tackle this instance.

Soft-Clipping:

This results from reads that only partially align against the reference sequence, such that bases at the ends of the read are considered non-aligned. This presents a problem because the SAM format starts the POS (position) count of the read at the first point of alignment.

Given the duplicate reads:

```
R1:AAATTGCGA
R1':AAGTTGCGA (sequencing error in third position)
```

We know just by looking at these that the reads are duplicates; however, if they align differently like so:

```
Ref:  AAATTGCGT
      |||||||||
R1:   AAATTGCGA

Ref:  AAATTGCGT
         |||||
R1':  AAGTTGCGA  (soft-clipping for first 3 bases)
```

The two reads will have completely different start positions! Thus, when assessing our reads for duplicates, we need to consider how to adjust for modified start positions due to soft-clipping.

UMIs

UMIs (Unique Molecular Identifiers) are known sequence tags that are appended onto reads to indicate the source library of the reads: a molecular barcode. We know that PCR duplicates, to be read from the same molecule of DNA, must all share the same UMI tags. However, if the UMIs have any sequencing errors or are read at poor quality, a base within the tag may be presented incorrectly. Thus, it's possible that duplicates can present with different sequences at the ends of the strands.

For this assignment, UMIs are assumed to be correct, and are not inline with the sequences in the sam file, but rather are included in the header of each sequence, at the end.

Algorithm Design:

Our first assumption is that we are starting with a uniquely mapped set of reads in SAM file format. This will be a requirement for out algorthim's input.

Sorting:

Next, we want to sort the SAM file such that all reads are sorted by leftmost (starting) coordinates (given in the 4th column of the file format). This will require a conversion of the SAM to BAM, and then performing the sort with samtools before converting back to SAM format.

```
>$ samtools view -b [in.sam] > [out.bam]
>$ samtools sort [out.bam] > [out.sorted.bam]
>$ samtools view -h [out.sorted.bam] > [out.sorted.sam]
```

Position Chunking:

Now that our reads are all sorted by position, we want to identify possible duplicates. To make sure we account for mismatching positions due to soft-clipping, each read needs to be compared to other reads that are within a *read length* of each other:

```
R1: AGATAGA
R1': TTATAGA

      123456789
Ref:  TGATAGATA
       ||||||
R1':  AGATAGA         Starting POS: 2

      123456789
Ref:  TGATAGATA
        |||||
R1:   TTATAGA     Starting POS: 3
```

These two are presumably duplicates, but don't share starting positions. We need to expand the search for possible dups up to the furthest distance that they can share alignments (which is one read length for single-end reads).

This offers the ability to 'chunk' our search for duplicates by only considering bundles of reads that have the same POS +/- Read length. *This will mean the untrimmed read length of the reads being processed must be a defined argument.*

Assuming reads are 40 bp long, an example of comparative chunking:

```
QNAME                         FLAG RNAME                       POS
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    65 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    77 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    98 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    99 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    100 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    131 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    200 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    220 ...
...

Chunk 1:
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    65 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    77 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    98 ...
```

```
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    99 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    100 ...

Chunk 2:
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    98 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    99 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    100 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    131 ...

Chunk 3:
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    200 ...
K00337:83:HJKJNBBXX:...:17333   83  CM001001.2/15519711-15520261    220 ...
```

Our program will therefore look at the position of each SAM line, then read the positions of subsequent lines until a read with a position outside the read length is found. The included reads will move onto the next step of processing and all other reads will be ignored for now.

Note, this has the added benefit of not requiring us to read all our reads into memory before processing.

Soft Clipping Position Adjustment:

The first read (with the lowest POS value) in each bundle will serve as the 'standard' which all other reads in the bundle are compared to (POS_{std}).

Starting with the next closest read in the bundle (by given SAM POS), get the CIGAR string from the 6th column and determine if the read has any soft-clipped bases. This is given by a numerical value followed by 'S' in the string, at the *start* of the string. If there are soft clipped bases, adjust the POS of that read by subtracting the number of clipped bases from that read's POS:

$$POS_{adj} = POS - S_{clipped}$$

If the POS_{adj} is equal to the POS_{std}, the reads will move onto the final check, to see if the UMI tags match. If the positions do not match, move on to the next read in the bundle (next closest POS to the first read) and repeat the process.

UMI Tag Match Check:

If two reads are found to be identical matches on alignment position (chromosome and sequence start position) the final check to determine if they are duplicates is to assess if the UMI tags match in the read headers (column 1). The UMI sequences are strings of nucleotides that are all 8 bases long included at the end of the header; when two reads are found to match position, the UMI's will be pulled from both headers and compared. If the strings match, the reads are definitive duplicates and move onto duplicate removal. If they don't match, the program moves on to compare the 'standard' read against the next read in the bundle.

Duplicate Removal:

In the instance that a duplicate is found, We want to throw out the *first* read of the duplicate pair (one with the smalled POS value). This will make sure that when we assess the next bundle of reads, we don't miss another duplicate.

Summary of Steps:

1. Convert sam -> bam, sort bam by POS, convert bam -> sam
2. Start at first read of SAM and grab POS (POS*)
3. Read in subsequent SAM lines that have POS values = POS* +/- Untrimmed Read length
4. Set POS_{std} = POS of first read in the bundle.
5. Get next closest read's POS and CIGAR fields. See if any soft-clipped reads exist at 5' end. Adjust POS value by clipped reads.

6. Compare POS_{std} to POS_{adj}. If equal, move onto UMI check.
7. Compare the UMI strings of both reads from the header. If the UMIs match, remove the *first* instance of the duplicate read, and move to next chunk.
8. If the clip-adjusted position does not match, or the UMI's do not match, begin comparison of the first read to the next read in the chunk, and repeat until duplicate is found or chunk is exhausted.

Pseudocode:

```
### Argparse arguments: take in -r (untrimmed read length) -f (sam file name)

## Function 1: Read Chunker
# Define Function that steps through sam file; start at read 1 and read in lines until
# POS of line is > -r

# Function 2: Position adjuster
# Takes output of Read Chunker as input
# Uses POS of first read and stores it as reference
# Checks CIGAR of each other read (for loop) and adjusts POS value if read has
# Soft-clipped bases at start of read (5' in context of the read)
# Compares adjusted/unadjusted POS values of other reads against POS of reference;
# if match, check if matching POS reads have matching UMIs; if these match, mark.
# At the end of the chunk, keep all non-duplicate reads, and the LAST read
# for any duplicates (with highest POS value) from marked reads

# Make function recursive, such that it calls itself with using the read set with
# the duplicates removed.

## Main Function:
# Import file contents as variable
# Call function 1 and Function 2
# Write out new SAM file with result of Function 2
```

Known Issues/Out of Scope:

Currenlty, there are a few situations where Duplicate reads could be missed and not removed. For completeness, I will list these issues here for transparency on what output can be expected from our de-duper.

1. *PCR duplicates differentially trimmed:* Given a set of PCR duplicates that have been trimmed to different read lengths, such that the alignmer may align the duplicates to the reference starting at differents left-most positions (POS), this method will not remove these duplicates. This is because the algorithm relies on matching position information after clipping correction, but is not trimmer aware.

2. *Paired-end reads:* Currently, this version of deduper is not designed to handle paired end reads; this will be addressed in future versions.

3. *"Bad" aligners:* In the case that the aligner used to create the SAM file aligns PCR duplicates to different reference regions, or mis-aligns reads due to a large number of sequencing errors, these duplicates will be unrecognizable to other, non-duplicate reads and thus will not be removed. This is not expected to be a problem based on the quality of the current alignment programs in popular use.

Summary of Examples:

1. Duplicates match exactly:

```
R1: ATGCGACT
R1': ATGCGACT

     123456789..
Ref: TTGCGACTG
      |||||||
R1:  ATGCGACT    POS: 2
R1': ATGCGACT    POS: 2
```

In this case, duplicate reads align exactly the same, have same sequence, and have same starting position.

2. Duplicates with offset by soft-clipped:

```
R1: AGGCGACT  (Seq error base 1)
R1': TGGCGACT

      123456789..
Ref:  TGGCGACT
       |||||||
R1:   AGGCGACT  POS: 2 (soft clipped by one)
R1':  TGGCGACT  POS: 1
```

In this example, duplicates will have different POS values in the SAM file, but R1's position will be adjusted based on the CIGAR string to match that of R1', and be removed.

3. Duplicates separated by other reads:

```
R1: ATTCGATGCA
R2: TTCGAGGCAT
R1': GTTGAATGCA (Sequencing error at 3,4 bp)

       123456789..
Ref:   ATTCGATGCAG
       ||||||||||
R1:    ATTCGATGCA    POS: 1
R2:     TTCGAGGCAT   POS: 2
R1':   GTTGAATGCA    POS: 6 (Called soft-clipped by 5)
```

This example just illustrates that soft-clipping can push the position (POS) of duplicates such that other, non-duplicate reads, may fall in-between them. This example illustrates why we cannot simply just check adjascent reads in the SAM file as duplicates.