Creating a De Novo Assembly Algorithm for Tn5 Transposase Tagmentation Reads

A Final Report for the Documentation and Potential Continuation of this Project

Joseph Yeh, Aug 21, 2024, Lan Lab, IBBME

Contents

Executive Summary

Although Tn5 transposase is widely distributed through Illumina's Nextera MiSeq library preparation kits, its resulting read libraries have not been assembled with the advantage of using the 9 base pair overlap. Because of this, most scientists continue to require as many initial copies of the genome as with the traditional ligase-based method of library creation. This means that for organisms that do not exist in populations of large enough samples for the Illumina Sequencer, they must be put through PCR to create enough copies of the genome, introducing unwanted errors and uneven coverage and relative amplification. Current assembly algorithms also create uncertainties for genomes with several repeating segments. Because of this, non-consecutive overlaps could be assembled by error, creating an incorrect order of fragments, and assembling an incorrect genome.

The hypothesis of this project is that by exploiting the overlapping 9 base pairs between fragments created by Tn5 transposase Tagmentation during assembly of the genome, less copies of the genome are required to create a library that can be assembled than using current assembly algorithms. The solution is an algorithm that considers the 9 base pair overlap between fragments to assemble fragment libraries, attempting to prove assembly can be done with fewer initial copies of genomes. This would minimize errors introduced by an initial PCR step which is currently required to create enough copies for a library to be assembled. These assemblies would also be more accurate as each consecutive fragment could be compared and assembled through the 9 base pair overlap, reducing the over or under coverage traditionally associated with assembly and preventing confusion caused by genomes with repeating segments.

The project found that this method of de novo assembly does reduce the number of initial cells required to produce higher quality contigs, though maybe only for a specific window of experimental parameters. Since less genomes are being used, the PCR step must be handled more carefully and have a more even relative amplification than conventional procedures to prevent under coverage of segments of the genome with a GC content below 10% or higher than 50%. This can be done by slowing down the rate of thermal cycling, although exact parameters are yet to be tested. It was also found that this method of assembly could create a comparable assembly to conventional methods with as few as 50 initial cells. However, this method also had an upper limit of initial cells as a population of 200 cells was unable to be assembled due to the amount of random overlap in the genome, the probability of random overlap increasing as the number of cells used increases. This gives a suspected window of use between 50 to ~150 initial cells with the optimal amount being 100 cells. Altogether, this method of assembly shows promise for the future of single cell analysis, particularly for those in the get microbiome.

Methods and Components of Project

The stages of this project are the creation of a Tn5 Tagmentation and read generation simulation, comparing simulated reads to experimental reads, creating an assembler specific to Tn5 Tagmentation, and comparing resulting contigs to conventionally created contigs. For the results in this project, the e coli genome was used as it is well known and has many available resources to compare results and generated statistics to real life data. As an example of a conventional assembly algorithm Spades is used as it includes error correction and is relatively robust in comparison to other assembly algorithms.

1. Simulation of Tn5 Tagmentation and Read Generation

To make it easy to alter parameters that different read library generation procedures could have, a simulation was made intending to resemble real life Tn5 Tagmentation and Illumina MiSeq read generation. From this, it could be determined which parameters were most optimal for assembly using the 9 base pair overlap and generate reads relatively quickly and at no cost. This also ensured that reads had a 9 base pair overlap as sometimes it is unknown if reads found through the Sequence Reads Archive (SRA) are Tagmentated using Tn5 Transposase. This simulation can be found under the file 'sim_linux.py' which takes in parameters specified in the README section of this document and outputs two Fasta files of equal number of entries as paired end reads, one for the reverse reads and one for the forward reads with the index of each entry in the Fasta file corresponding to the same fragment read.

Transposase Tagmentation:

The simulation first splits the input genome relatively randomly into fragments of between 150 and 10,000 base pairs long along a skewed normal distribution such that the mean fragment length is approximately 500 base pairs. To achieve this, the skewness factor parameter for the python library function 'scipy.stats.skewnorm()' was set to 3. To incorporate an approximation of Tn5's insertion bias, splits are twice as likely to happen at locations where either the base pair afterwards is a G or the ninth base pair from the prospective split is a C. These splits are then given the 9 base pair overlap by saving them to a list in the order they appear in the genome and putting the last 9 base pairs of the previous sequence in the beginning of each element of the list. For the first sequence of the list, the last 9 base pairs of the genome are used as bacterial genomes are circular. The reverse complement of each sequence in the list is then appended to the list of originally created fragments. The simulation then adds P5 or P7 adapters to the 5-prime end of each fragment, and the reverse complement of either adapter to the 3-prime end. This is randomly done to simulate the chance that a fragment will be readable by the flow cell. The above process is done as many times as there are genomes available (given as a parameter to the simulation), each run creating different sequences and splits at different positions as the process is relatively random each time.

Read Generation:

The simulation first gets rid of all unreadable fragments that do not have different adapters on either side of the fragment as fragments with the same adapter on either side are unreadable to Illumina Miseq technology. This is roughly 50% of the generated fragments. The fragments are then put through PCR based on the GC% and the PCR relative amount plateau which is taken as a parameter to the simulation. The plateau can be controlled in real life experiments by changing the rate of heating and cooling by the thermal cycler. The output of the PCR stage is a python dictionary with the fragments as keys and the number of copies of each fragment as their respective values. At this point, all the fragments in the dictionary are readable. The simulation then shuffles the dictionary before randomly picking a specified number of fragments to be read. The number of fragments read is given as the parameter of flow cell size. During reading, the simulation identifies and removes the adapters and adapters' complements, then reads 250 base pairs on both the 5-prime and 3-prime ends of each selected fragment. The error rate of Illumina Miseq is considered with the probabilities that a substitution, insertion, or deletion will occur at each base (each of which is twice as likely to occur in the reverse read). This creates two lists, a forward and reverse list, who's elements at the same indices correspond to the same fragment. For example, the first entry of the forward read list is the 5-prime read of the fragment that produced the first entry of the reverse read list (which is the 3-prime read of that fragment). It is important to note that if the fragment was less than 250 base pairs long, the whole fragment would be appended to both the forward and reverse read lists. Each list is then saved to its own Fasta file as specified by the user in the simulation parameters.

2. Analysis of Simulated Read Library and Comparison with Real Life Experiment Read Libraries

To be able to evaluate the reliability of the simulation of creating realistic paired end read libraries, reads of the simulation were compared to reads from the Sequence Read Archive (SRA). The same parameters as conventionally used were input into the simulation (such beginning with thousands of genomes, conducting 35 cycles of PCR, etc.) and then those simulated reads were put into Spades and Bowtie2 to get metrics of their coverage and quality. The methods for obtaining these metrics can be found in the file 'file_processing.py'. Then, an e coli read from an SRA was used to get the same metrics through spades and bowtie2.

3. Creating an Algorithm for Assembling Reads by 9 base pair Overlap Created by Tn5 Tagmentation

After confirming that the reads from the simulation were an approximate representation of real experimental paired end reads, an assembler that considered the overlapping 9 base pairs from the original Tn5 Tagmentation was created. This program can be found in the file 'assembler_dict.py'. The program takes the file location of the forward and reverse read Fasta files as well as other file parameters as specified in the README section as input, and outputs an intermediary dictionary used for hashing the reads as well as a scaffold of assembled reads by their overlapping 9 base pairs.

The program first opens the forward and reverse read Fasta files, putting each in its own respective list. Here, the element at each index of each list corresponds to the same fragment as before in the output of the simulation. The forward and reverse reads are then compressed into one python read list with the format of each element being "forward read + 'NNNNNNNNN' + reverse read". The simulation should have created an equal amount of forward and reverse reads, allowing this to work. Since transposase creates fragments with an average length of 500 base pairs and the forward and reverse reads are on average 250 base pairs in length, inserting 9 unknown bases in between the reads approximately estimates the gap between the fragments as 9 base pairs are overlapped and are not accounted for in the reads. Although gaps may be larger than this or may not exist at all as the reads overlap with a fragment shorter than 500 base pairs, this approximation is enough for modern assemblers to make assumptions from and perform error corrections.

Using the compressed read list, the program then creates a python dictionary with the keys as the first nine base pairs of the forward read and the value being a list of all fragments that begin with the key of 9 base pairs. This allows the program to keep repeated fragments which is essential for Spades later during error correction. Another dictionary is then made using the same logic, but instead using the last 9 base pairs of the reverse read as the key this time. The result is 2 dictionaries, each with all the reads, but hashed differently. We will call them the forward and reverse dictionaries respectively. These dictionaries are saved into two 'txt' files to document the hashing of the reads.

The program then begins assembling reads into scaffolds by first iterating through the values in the forward dictionary, searching for either an exact key match in the reverse dictionary or a reverse complementary match in the forward dictionary. If a match is found, the first element of the value list of each read are assembled accordingly and these reads are removed from the dictionaries. The assembled scaffold is then appended to a python list. Afterwards, the program iterates through the reverse dictionary searching for only a reverse complementary match in the reverse dictionary. It does not search the forward dictionary as any matches would have previously been assembled when iterating through the forward dictionary. After matching, the assembled scaffolding is then appended to the scaffold python list from before.

The assembler then repeats the whole dictionary creation and scaffold assembly process, this time using the scaffold list instead of the read list to create the dictionaries, until the returned scaffold list is not shorter than the starting scaffold list. Essentially, a while loop until no more scaffolds can be assembled and no more matches can be found.

Once the while loop has ended, the resulting scaffolds are saved as a Fasta file to the location of your choice.

4. Comparison between Conventional Assembly Algorithms to 9 base pair Overlap Assembly

To compare conventional assembly to assembly when using the preassembled by 9 base pair scaffolds, the scaffolds made by the Tn5 specific assembler were input to Spades as '—trusted contigs' as well as the original reads created by the simulation. This allowed Spades to conduct error correction on both the reads and the scaffolds, ensuring they lined up and not relying on any single one. The resulting contigs were then compared to the ones created by Spades using the SRA reads and the simulated reads alone. Metrics used for this comparison were the number of created contigs, the range of contig lengths, and average length of the contigs.

Results

Simulation Results

When assessing the coverage of simulated reads compared to the SRA reads, it became clear that two important parameters were the plateau of the PCR step (determined by the rate of heating and cooling of the thermal cycler) and the number of genomes the simulator began with. Although conventionally 35 cycles of PCR are used with a plateau of approximately 10-50%GC, previously done experiments have lengthened this plateau to as far as 11-84%GC, requiring as few as 16 cycles of PCR to achieve enough genomic material for the sequencer which is 20 nanograms. Having a larger plateau range with less cycles of PCR allows a more even amplification of fragments to increase coverage of carrying GC content regions.

From the figures below, the simulation provides a good approximation of experimental reads and it was decided to continue with 16 cycles of PCR at a plateau of 11-84%GC.

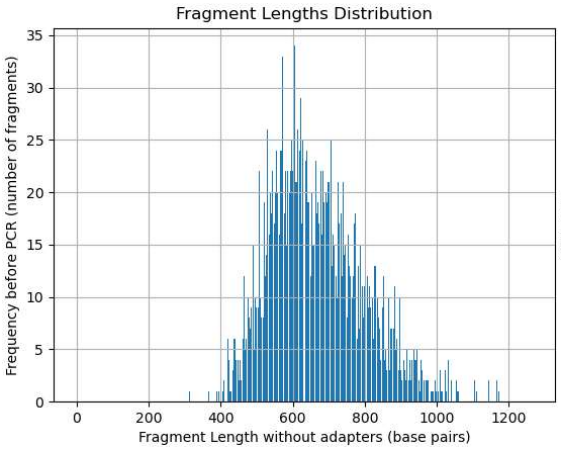


Figure 1: Fragment length distribution of Tn5 along 1 genome. Shows an accurately skewed normal curve with a mean length of 500 bp.

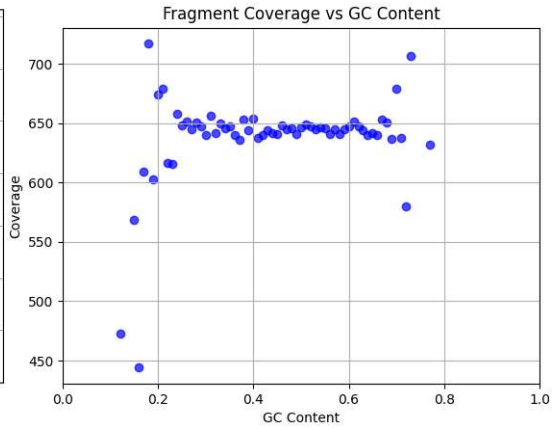


Figure 2: GC% compared to amount of times the segment of the genome was covered. This demonstrates the effect of the plateau range.

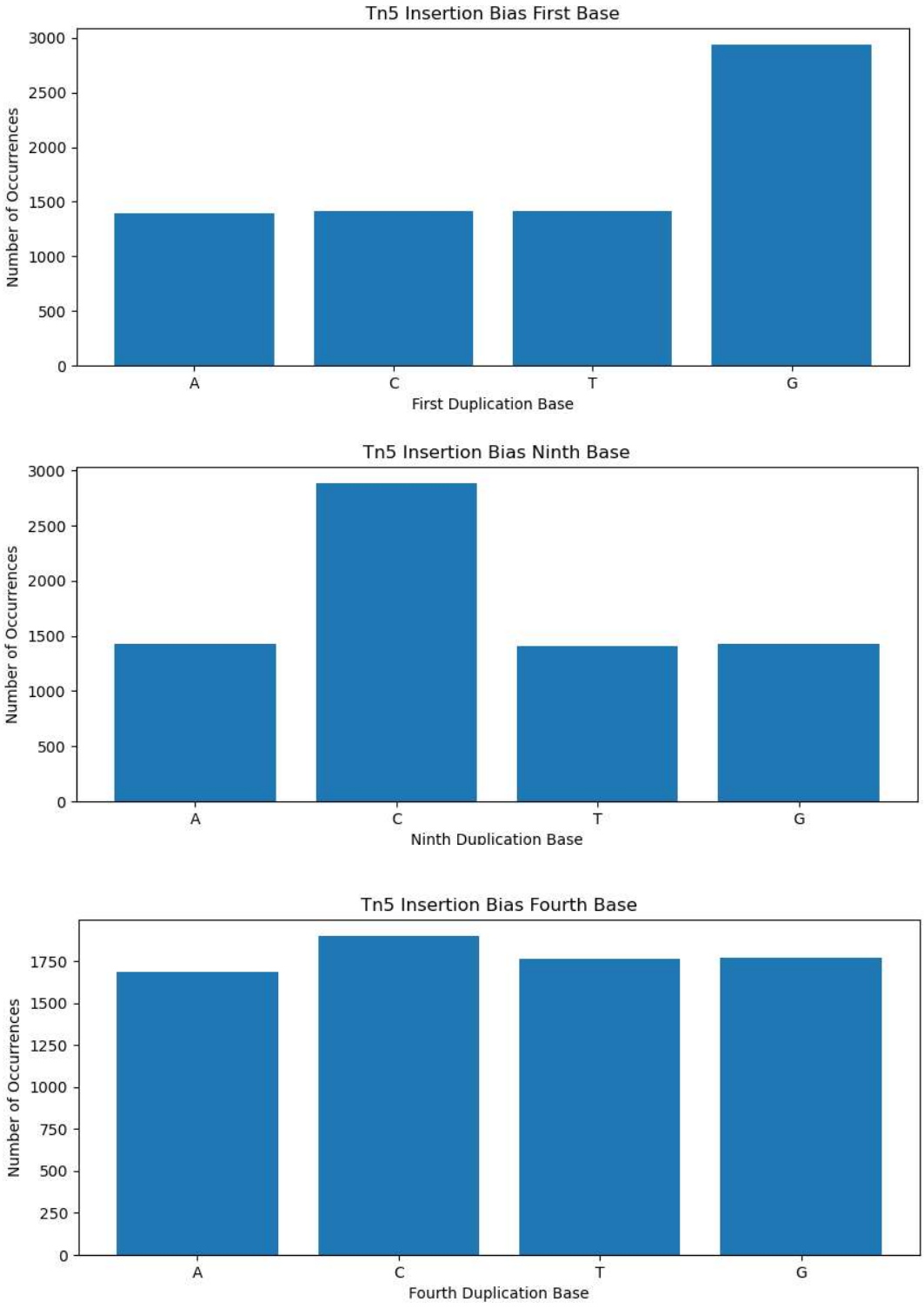


Figure 3: Number of times each base appears at the first, ninth, and fourth base position from the location of the fragmented split. This demonstrates the simulation's ability to replicate the Tn5 insertion bias. The fourth base pair position is shown as a comparison to a position with minimal impact on Tn5 insertion bias.

Below are coverage histograms of different reads with how many times a section of the genome is covered on the x-axis and the number of sections covered that number of times on the y-axis. These were used as comparisons to decide which parameters to use in the simulation.

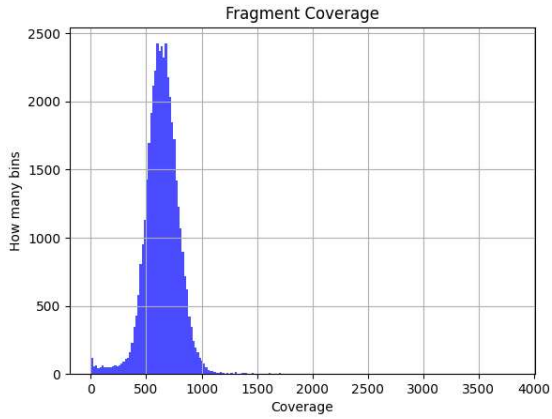


Figure 4: SRA read coverage

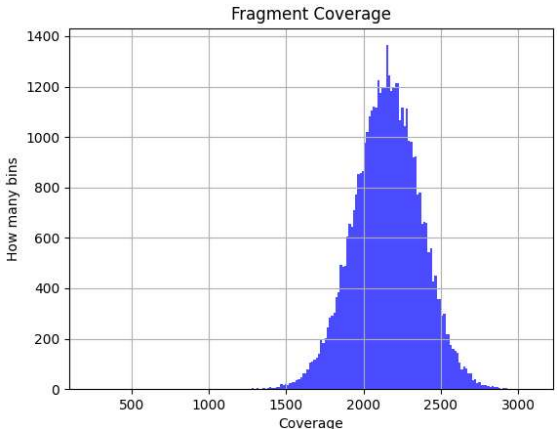


Figure 5: Simulated read coverage at 1000 genomes and 35 cycles of PCR with PCR bias of 10-50%

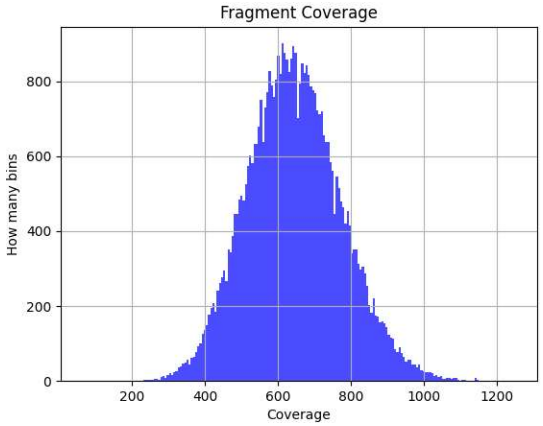


Figure 6: Simulated read coverage at 100 genomes, 16 cycles of PCR, but no PCR biases

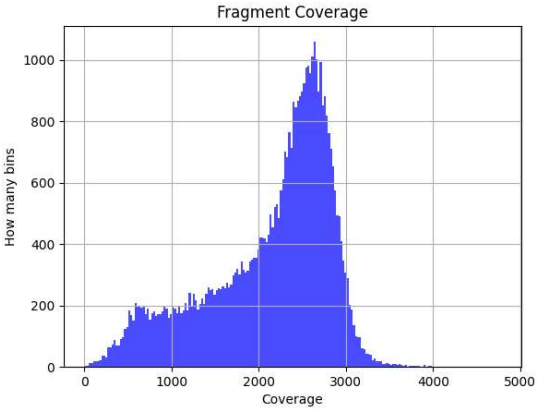


Figure 7: Simulated read coverage at 100 genomes, 16 cycles of PCR, with PCR plateau of 11-84%

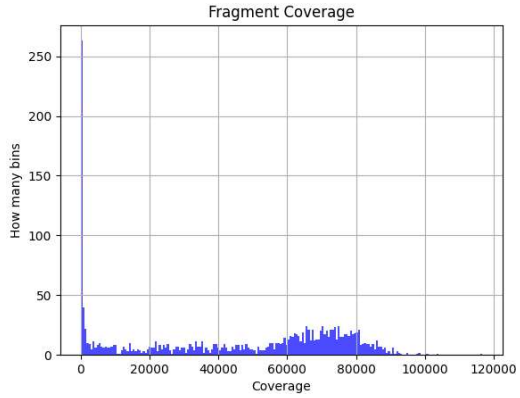


Figure 8: Simulated read coverage at 100 genomes, 16 cycles of PCR, PCR plateau of 10-50%.

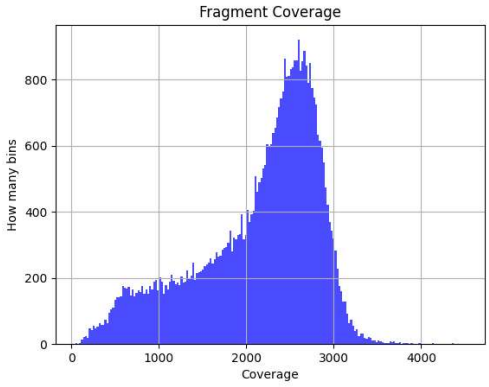


Figure 9: Simulated Read Coverage at 100 genomes, 35 cycles of PCR, PCR plateau of 11-84%.

Assembly Results

Table 1: Below is a table of the different reads that were assembled and the characteristics of the contigs that resulted when they were run through Spades. Also included is whether they were pre-assembled using the 9 bp assembler for Tn5 reads which can be used to assess the effectiveness of the novel assembly method. All the simulated reads use 16 cycles of PCR at a plateau of 11-84%GC.

Read Assembled	**Using 9bp Assembler? (yes / no)**	**Number of Resulting Contigs**	**Length Range of Contigs (bp)**	**Average Contig Length (bp)**
SRA read	no	2156	130 – 67,000	1,855
Simulated: 100 genomes	no	326	130 – 11.000	12,269
Simulated: 5 genomes	yes	1931	150 – 25,000	2,071
Simulated: 1 genome	yes	1076	130 – 11,000	3,717
Simulated: 100 genomes	yes	66	130 – 550,000	60,606
Simulated: 50 genomes	yes	176	130 – 640,000	22,727
Simulated: 200 genomes*	yes	N/A	N/A	N/A

**was not able to get data because would cause error in Spades when ran.*

```
Seq('GATGTTACC'):
[Seq('GATGTTACCGGCGGGCTGCAATTTACTTACATATCACTGGATGATTACCGCATT...GCA'),
Seq('GATGTTACCCGTCGTGGCGTTAAGCAAGTGAAAGGGTTTGGCGACCATCTGACC...CTG'),
Seq('GATGTTACCCGTCGTGGCGTTAAGCAAGTGAAAGGGTTTGGCGACCATCTGACC...CTG'),
Seq('GATGTTACCTTCCTGAATCAAATCCGCCTGTGGCAGGCCATAGCCCGCATAATT...GGC'),
Seq('GATGTTACCTATTCGTCGTGATAGCCCGAAAAAGGCAATTACCTTGATTCTGGC...GCA'),
Seq('GATGTTACCGGCGGGCTGCAATTTACTTACATATCACTGGATGATTACCGCATT...GCA')]

Seq('TATTACGCA'):
[Seq('TATTACGCAGTGCAAAAGTGGTATCGGAATGATTCGTGTATTCCTGCATAACAG...ATC'),
Seq('TATTACGCAGCCCTTCGGGTGTCTCTTTGCTCCCCTTCGCTACGGGTTTATATC...TGA'),
Seq('TATTACGCAGCCCTTCGGGTGTCTCTTTGCTCCCCTTCGCTACGGGTTTATATC...TGA'),
Seq('TATTACGCAGTGCAAAAGTGGTATCGGAATGATTCGTGTATTCCTGCATAACAG...ATC'),
Seq('TATTACGCAGCCCTTCGGGTGTCTCTTTGCTCCCCTTCGCTACGGGTTTATATC...TGA')]

Seq('CCACAATTC'):
[Seq('CCACAATTCTAGTAGTGAGCCAGGGAAAACGACAGCCAGCGCGTCGCCCTGACC...TGC'),
Seq('CCACAATTCTAGTAGTGAGCCAGGGAAAACGACAGCCAGCGCGTCGCCCTGACC...TGC'),
Seq('CCACAATTCCCGGCGTATGCAGCGGCTCGGCGTTGCCCAGCGTTTCGATAATCA...TTG')]

Seq('ATTCCCAGG'):
[Seq('ATTCCCAGGCTTCCTGTTTCGCCAGGACAGCCTGCAGCAGTTTGTTATTCAGTG...CGG'),
Seq('ATTCCCAGGGCGCTACAGGACTGCCACTGACCTTCCGGGATCGCACGAATTGCA...GGT'),
Seq('ATTCCCAGGGATATTGTTGCCAACCATACCAGACAATTCGATCATCAATATTTA...TTT'),
Seq('ATTCCCAGGCTTCCTGTTTCGCCAGGACAGCCTGCAGCAGTTTGTTATTCAGTG...CGG'),
Seq('ATTCCCAGGGATATTGTTGCCAACCATACCAGACAATTCGATCATCAATATTTA...TTT'),
Seq('ATTCCCAGGCTTCCTGTTTCGCCAGGACAGCCTGCAGCAGTTTGTTATTCAGTG...CGG'),
Seq('ATTCCCAGGGCGCTACAGGACTGCCACTGACCCTCCGGGATCGCACGAATTGCA...GGT')]
```

Figure 10: Shown are examples of the key-value pairs present when hashing the reads in the 9bp assembler. This particular dictionary is the forward dictionary for the simulated reads using 100 genomes used in table 1. Keys are not unique to specific fragments as many fragments can share the same first 9bp. This is statistically probable as there are only 262144 different combinations of 9bp keys and Miseq samples 2,000,000 reads.

Since the more genomes used by the simulator the more likely it was that several different fragments would share the pattern of overlapping 9bp, it was at initially thought that using less genomes would increase the assembler's ability to create scaffolds and avoid assembling incorrect reads together. However, decreasing the number of genomes used for creating reads also decreases the diversity in split locations of Tn5, decreasing the number of 9bp overlaps and overall coverage. This lack of not just 9bp overlaps, but random overlaps as well creates an issue for both the 9bp assembler and conventional assembly algorithms. Here, a tradeoff had to be discussed: more initial genomes resulted in a higher probability the resulting scaffolds would be false and not actually exist within the genome, while less genomes decreased the amount of overlaps overall and could potentially leave segments of the genome uncovered.

This issue was later mitigated when running the assembled scaffolds from the 9bp assembler through Spades as '—trusted contigs'. When the simulated scaffolds from non-uniquely hashed reads were assembled using Spades, it resulted in fewer contigs that were significantly longer and did in fact exist in the e coli genome (though often with singular base pair errors from the reading of the fragments). Since Spades incorporated error correction and would trim edges in its de Bruijn graph that were not repeated, it is assumed it simply got rid of scaffolds that contradicted each other that were created by the non-unique hashing made when using 50 or more initial genomes. A hypothesis on why the 200 genome simulated reads would cause errors is that it had too many non-unique 9bp keys for the assembler to do error correction and would cause the algorithm to be unable to finish assembly.

The conclusion that can be drawn from these results is that this method of assembly should be investigated further to validate these results and find the window of parameters for which this method can be applied. This method shows promise for single-cell analysis and producing higher quality de novo assemblies for unknown bacterial genomes that may not exist in large population sizes or mutate quickly.

Next Steps

Immediate next steps to further investigate this method of assembly would be to pass simulated reads made from even more varying amounts of initial cell populations (below 50 and larger than 100) to gauge where this method of assembly ceases to be more effective than current assembly algorithms alone or begins to create false contigs. It would also be beneficial to investigate further how differing the number of cycles of PCR and different plateaus would impact assembly as so far only read coverage has been assessed.

Further metrics to investigate would be mapping the coverage and errors produced in assembled contigs to a reference genome to gauge accuracy. An N50 metric to better understand the contig length and assembly efficacy would also be required.

README

Below are instructions to be able to use the full pipeline from read simulation to getting assembled contigs. If wanted, the simulation may also be used on its own to experiment with various read generation parameters before conducting experiments. Further information used to create the simulation and assembler can be found in the file 'file_logs.txt'.

Tn5 Tagmentation and Read Simulator

The simulator can be found as the file 'sim_linux.py'. A backup of it is found as 'sim_backup.py'. This will take in the inputs specified in the parameters below and output pair ended reads with the forward and reverse reads each having their own Fasta files.

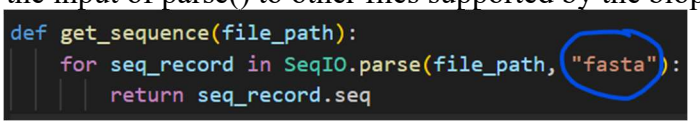
Under the main() function, you will find a list of all easily alterable parameters:

```
def main():

  """#Implementation library creation + reading"""
  start = time.time()
  #parameters
  verbose = False      #print figures and stats or not
  genome = get_sequence('/home/jyeh/summer2024/ecoli.fasta')  #input initial genome
#   genome = plasmid_seq
  num_genomes = 200   #change this number for how many genomes there are; 100 is best for our application
  adapters = ['AATGATACGGCGACCACCGA',  'CAAGCAGAAGACGGCATACGAGAT'] #adapters used for transposase tagmentation
  plateau = [11, 84]  #pcr plateau range depending on protocol
  cycles_of_pcr = 16   #number of cycles of pcr
  num_flowcells = 1   #number of flowcells used for reading (will multiply number by 96 for batch size)
  read_length = 250   #length of reads determined by machine
  flowcell_size = 2000000  #number of samples per flowcell; miseq has 25 million reads per flowcell, micro has
  fwd_filename = '/home/jyeh/summer2024/test_reads.fasta'
  rvs_filename = '/home/jyeh/summer2024/test_reads.fasta'
```

List of parameters:

- Verbose:
  - if True, will print intermediary figures and fragment / read statistics
    - stats include:
      - fragment lengths
      - errors produced
      - Tn5 insertion Bias per duplication position
      - Number of readable fragments at each stage of reading
      - Time for each stage to run
  - if False, will not

- Genome:
    - Path to the genome as a Fasta file
    - To change file type of input genome, can alter the get_sequence() function and change the input of parse() to other files supported by the biopython library:
        - 
```
def get_sequence(file_path):
    for seq_record in SeqIO.parse(file_path, "fasta"):
        return seq_record.seq
```
- Num_genomes:
    - Cell population size that Tn5 Tagmentation is being applied to
- Adapters:
    - A list as [P5 adapter, P7 adapter]
- Plateau:
    - Plateau range of PCR procedure determined experimentally by heating and cooling rate of thermocycler
    - As format [start of plateau range, end of plateau range]
- Cycles_of_pcr:
    - Number of cycles of PCR for simulation to perform
- Num_flowcells:
    - Number of flow cells being used to read fragments
    - Determines the number of reads generated
- Read_length:
    - Length of reads generated by reader
    - May vary by +-2 bp because of insertion of deletion errors
- Flowcell_size:
    - Number of fragments on each flowcell
    - Will produce this many forward reads and this many reverse reads per flowcell
    - Total number of reads = flowcell_size*2*num_flowcells
- Fwd_filename:
    - Where forward reads will be stored. Must be a Fasta file.
- Rvs_filename:
    - Where reverse reads will be stored. Must be a Fasta file.

Advanced parameters to change fragment sizes created by Tn5:

These can be found under the splitter() function in the script.

They are shown in the dark blue box below and change the skewed normal distribution to which fragments lengths are fit to.

```
#breakup sequences into multiple random segments
def splitter(sequence, verbose = False):
    res = []
    splits = []
    length  = len(sequence)
    split = 0

    a = 3  # Skewness parameter
    loc = 500  # Location parameter (mean)
    scale = 200  # Scale parameter (standard deviation approximation)
    min_length = 150 #minimum transposon length
    max_length = 10000 # max length of transposon
```

File processing

To obtain metrics such as error rate from Sam files, the coverage graphs, the starting point of fragments, or relative amount of PCR amplification, a separate python script was used. This can be found in ‘file_processing.py’ or backed up as ‘file_processing_backup.py’.

To convert Fasta file to Fastq file (as Spades only takes Fastq files for reads) change the parameters in the ‘’’convert fasta to fastq’’’ section and then run:

```
'''convert fasta to fastq'''
start = time.time()
fwd_fasta_file = '/home/jyeh/summer2024/genomes_200_fwd_reads.fasta'
fwd_fastq_file = '/home/jyeh/summer2024/genomes_200_fwd_reads.fastq'
fasta_to_fastq(fwd_fasta_file, fwd_fastq_file)
rvs_fasta_file = '/home/jyeh/summer2024/genomes_200_rvs_reads.fasta'
rvs_fastq_file = '/home/jyeh/summer2024/genomes_200_rvs_reads.fastq'
fasta_to_fastq(rvs_fasta_file, rvs_fastq_file)
end = time.time()
print(f"Time taken: {end-start}")
```

To analyze hashed dictionaries from the 9bp assembler, use and change parameters in the ‘’’analyzing assembled dicts’’’ section:

```
'''analyzing assmebled dicts'''
fwd_dict_file = '/home/jyeh/summer2024/genomes_100_fwd_dict.txt'
rvs_dict_file = '/home/jyeh/summer2024/genomes_100_rvs_dict.txt'
fwd_dict, rvs_dict = open_dict_txt(fwd_dict_file, rvs_dict_file)
print(f"Number of fwd keys: {len(fwd_dict)}", f"Number of rvs keys: {len(rvs_dict)}")
unique_seq_counter(fwd_dict, rvs_dict)
```

To get coverage metrics from a Sam file change the samfile_path parameter in the ‘’’coverage visualization’’’ section:

```
'''coverage visualization'''

samfile_path = "/home/jyeh/summer2024/no_pcr_.sam"

'''plot error rate and alignment position biases'''
start = time.time()
plot_error_and_fragment_bias(samfile_path)
# plot_qual_scores_distribution(samfile_path)
plot_fragment_coverage(samfile_path)
print('time:', time.time() - start)
plot_error_distribution(samfile_path)
end = time.time()
print(f"Time taken: {end-start}")
```

Bowtie2 and Spades Commands

These are algorithms that are publicly available that were used for the project. You must ensure they are installed via bioconda before running their commands and that you are using a linux OS. The number of threads you are using can be found by running the following line in a script: *print('cpus:', multiprocessing.cpu_count())*

Bowtie

To map the reads and contigs and obtain further metrics, bowtie 2 was used. Below is the format of the command lines used:

1. Build the reference genome:
    a. *bowtie2-build path_to_reference_genome.fasta path_for_reference_to_be_placed*
2. Map Reads
    a. For paired end reads:
        i. *bowtie2 -f -p num_threads -x path_of_reference_placed_from_above -1 path_of_forward_reads.fasta -2 path_of_reverse_read.fasta -S path_for_output_sam_file.sam*
    b. For single reads or contigs:
        i. *bowtie2 -f -p num_threads -x path_of_reference_placed_from_above -U path_of_contig_or_read.fasta -S path_for_output_sam_file.sam*

Spades

To assemble the reads or scaffolds, Spades was used. Be sure that the reads are converted from Fasta files to Fastq files first if using simulated reads. Spades only takes Fastq files as reads. A conversion script can be found in 'file_processing.py'. Below is the format of the command lines used:

- For paired end reads:
    - *spades.py -1 path_to_forward_reads.fastq -2 path_to_reverse_reads.fastq -o folder_name_for_spades_output -t num_threads –phred-offset 64*
- For single end reads:
    - *spades.py -s path_to_single_end_read.fastq -o folder_name_for_spades_output -t num_threads –phred-offset 64*
- For assembling reads using pre-assembled scaffolds (like from the 9bp assembler):
    - *spades.py –trusted-contigs path_to_scaffold_or_contigs,fasta -1 path_to_forward_reads.fastq -2 path_to_reverse_reads.fastq -o folder_name_for_spades_output -t num_threads –phred-offset 64*
- phred offset is set to 64 because it is for illumina paired end reads. Sanger sequencing would use a phred offset of 33

Using the 9bp Assembler:

The script for the 9bp Assembler can be found in the file ‘assembler_dict.py’. It takes in the following parameters and outputs the forward and reverse hashed dictionaries in txt files as well as scaffolds assembled from paired end reads using their 9bp overlap.

List of Parameters:

```
if __name__ == '__main__':
    fwd_fasta = '/home/jyeh/summer2024/genomes_200_fwd_reads.fasta'
    rvs_fasta = '/home/jyeh/summer2024/genomes_200_rvs_reads.fasta'
    output_file = '/home/jyeh/summer2024/assembler_dict_scaffold_200_genomes.fasta'
    fwd_dict_file = '/home/jyeh/summer2024/genomes_200_fwd_dict.txt'
    rvs_dict_file = '/home/jyeh/summer2024/genomes_200_rvs_dict.txt'
```

- Fwd_fasta:
  - The path to your forward reads file as a Fasta file
- Rvs_fasta:
  - The path to your reverse reads file as a Fasta file
- Output_file:
  - The path the scaffolds will be placed in. Must be a Fasta file
- Fwd_dict_file:
  - The path your forward hashed reads will be saved to
  - These are hashed by the first 9bp of the forward read
  - Must be a txt file
- Rvs_dict_file:
  - The path you reverse hashed reads will be saved to
  - These are hashed by the last 9bp of the reverse read
  - Must be a txt file