Error detection in *de-novo* sequence assembly

Ricardo Augusto López Torres
ricardo.lopez16@upr.edu
Computer Science Department
University of Puerto Rico - Río Piedras

Advisor:
Humberto Ortiz-Zuazaga
humberto.ortiz@upr.edu
Computer Science Department
University of Puerto Rico - Río Piedras

Abstract

Knowledge of an organism's DNA sequence is instrumental in understanding its fundamental biological properties. In order to discover an organism's DNA sequence, DNA sequencing and *de-novo* sequence assembly are necessary. In this work, we present a method for doing *de-novo* sequence assembly and a method for error detection in DNA sequencing data. We test these methods on simulated sequencing data and on real sequencing data from the Drosophila Melanogaster genome. We show that the error detection method is effective with simulated data, and the assembly method is effective with both simulated and real data.

1 Introduction

In biology, **DNA sequencing** allows us to get information on the nucleotide structure, or **sequence**, of a sample of DNA. A **DNA sequencer** is an instrument that receives a sample of DNA and outputs many small fragments, or **reads**, of its sequence. **Sequence assembly** is the process by which we join these fragments in order to reconstruct the complete sequence. ***De-novo***, meaning "from the beginning", refers to sequence assembly done without a reference genome, and it is used when trying to discover new genome sequences.

A complication in sequence assembly stems from possible errors in the sequencing data. The sequencer may output reads with one or several mismatches from the original sequence. Using these erroneous reads in the reconstruction of the DNA sequence can lead to assembling incorrect sequences. In *de-novo* sequence assembly it becomes particularly challenging, due to not having any reference to compare with and verify the integrity of the reads.

In this work, we present an implementation of a de-novo sequence assembly method for reconstructing DNA sequencing data. We also present a method for detecting errors in sequencing data.

2 Methodology

2.1 Sequence assembly

The method we present for sequence assembly on the reads produced by a DNA sequencer has three fundamental steps:

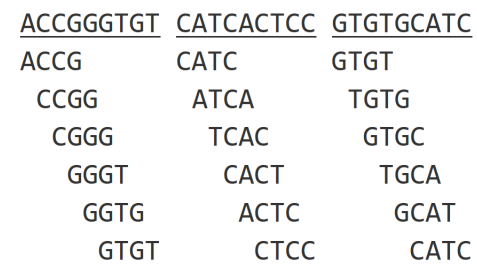
Step 1 For some k, split each read into its k-mers.
A **k-mer** is a k-length substring.

Step 2 Construct a directed graph over the k-mers, placing a directed edge between k-mers that overlap.

Step 3 Find an Eulerian path in the graph.
An **Eulerian path** is a path in a graph that visits every edge exactly once.

ACCGGGTGT CATCACTCC GTGTGCATC

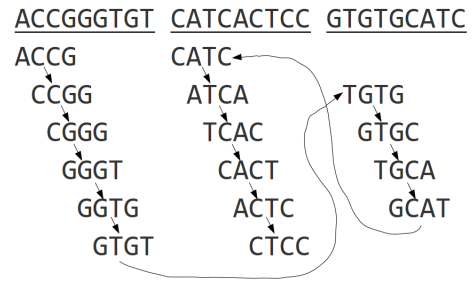
Reads to be assembled



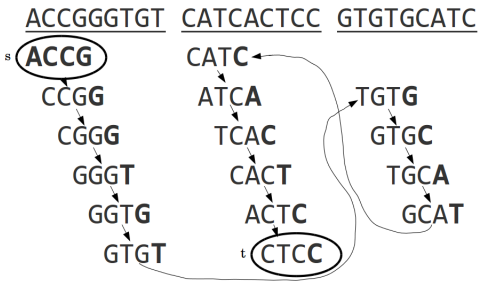
Step 1 Reads split up into k-mers, $k = 4$

2.2 Error detection

One possibility for detecting sequencing errors is through the use of k-mer counting and abundance filtering. This is done by pre-processing the data before assembly, counting how many times each unique k-mer occurs in the data and filtering the dataset by eliminating from the reads the k-mers that occur fewer times, or have low abundance. These low-abundance k-mers either correspond to sections of the sequence that had low coverage, or possibly to errors in the sequencing data. Eliminating them will then, ideally, eliminate all the errors from our dataset.



Step 2 Directed graph constructed over k-mers



ACCGGGTGTGCATCACTCC

Step 3 Assembly of Eulerian path, from s to t

2.3 Implementation

We wrote our assembler **assemble.py**[2] in Python 2.7. We're using NetworkX[3] to represent our k-mer graph, khmer[4] for k-mer counting and filtering, and screed[5] for parsing input FASTA and FASTQ read data.

We implemented the **find_eulerian_path**[6] function for finding an Eulerian path in a NetworkX graph. Initially, the **find_eulerian_path** function was incomplete and would not detect if a graph did not have an Eulerian path in it and would produce incorrect sequences for those graphs.

Later on, we were able to improve our **find_eulerian_path** function, as discussed in [6], to correctly find eulerian paths in directed graphs, and to raise an exception when the graph did not have an eulerian path. In this way, if an erroneous k-mer graph was produced we will ignore it instead of producing erroneous sequence.

Our implementation of **assemble.py**, when working with a dataset of ∼1.3 million reads, consumes ∼4GB of memory and executes in ∼3-4 minutes running on an Intel Core i7-3610QM CPU @ 2.30GHz.

3 Work

Our first step was to check how k-mer counting could help us detect errors in the sequencing process using a simulated dataset. We generated a random DNA sequence of length 1000, and chose 500 positions on the sequence to make reads of length 35. We inserted errors into 15 reads, in order to simulate a 3% error rate. To insert an error, we chose a random base in the read swapped it with a different, "incorrect" one. We saved these generated reads in FASTA format, annotating the error reads accordingly (See Figure 1). Then, we ran the dataset with errors through **assemble.py**, but because we still had not fully implemented **find_eulerian_path**, **assemble.py** was producing incorrect sequences.

```
>481|368
aataaaatcaagtgtgggtttgtaggtctcgttga
>482|732
cattcccaacagaaggctccgatacgattcaaatc
>483|223
agatgaagtctatcaccttaggctcaaggtttagg
>484|904
ccaatggacttaatatgaatggcagctgataacaa
>485|717|error
cgacaaaTatgtctccattcccaacagaaggctcc
>486|464|error
taatcttggaaGttgccacgttcatgacgattgaa
>487|850|error
gctcctcggttcaagtccggcgcAggaagaaagaa
```

Figure 1: Generated reads, with inserted errors marked in uppercase letters

Next, we tried to use abundance filtering to clean up our dataset. We used the khmer k-mer counting script **load-into-counting.py**[4, 7, 8] to create a k-mer counting table for this read dataset with $k = 9$, and used the abundance filtering script **filter-abund.py**[4, 7] to remove the low-abundance k-mers. This script produced a new FASTA read file with the filtered reads. With our naive example, filtering out k-mers that ocurred only one time was enough to remove all of the simulated errors we had inserted into the reads (See Figure 6). Using the filtered dataset with our assembler, we were able to reconstruct 99.4% of the original sequence with 0 errors (see Figure 2).

At this point, we moved on to testing our method on real data from Drosophila Melanogaster genome sequence spike-in experiment GSE20555[9], which contained reads of length 36 spanning the entire Drosophila genome. We limited ourselves to data from the GSM516588 library run, which contained ∼13 million reads. The experiment data included a map file GSM516588_run32_s_1_ERCC_dm3_map.txt (See Figure 3) for reads with two or fewer mismatches that had the aligned read sequences and additional information on that read, such as which genome the read mapped to, its location on the genome, and which positions in the read contain errors, if any.

First, we wanted to see what kind of output **assemble.py** produced with clean reads from real data. We started by only looking at clean (error-free) reads from the 2L genome. Using the map file information, we extracted the clean

Figure 2: Output of our assembler with filtered reads from simulated data.

Figure 3: Map file including additional information for each read

reads that mapped to the 2L genome from the map file, which amounted to ∼1.3 million reads, and saved them in FASTQ format. The reads in the dataset can be either forward or reverse aligned, but the reads in the map file have all been corrected to be forward aligned, which simplifies the assembly method.

We then ran the clean 2L reads through **assemble.py**. By experimenting with different values for k, we found that the k values that produced the best output for this dataset were $k = 18$ and $k = 20$, which roughly correspond to half the length of the reads, which were of length 36. With $k = 18$, our assembler produced 50,507 total contigs. Of those, 312 contigs had length $\geq$ 500 (see Figure 4), which likely represent the areas of the genome that had good coverage. The longest sequence produced had length 5412.

We looked at the 312 contigs with length ≥ 500 and passed them through BLAST[10] to verify that the assembler output was correct (see Figure 5). BLAST is a database containing all currently known genomes that allows us to query if a sequence we produced matches with an existing reference genome. We compared our produced sequences against the Drosophila Melanogaster reference genome and found that 311 of the 312 sequences produced made a %100 match to the reference genome.

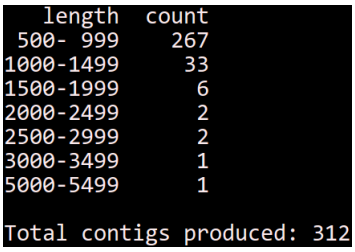


Figure 4: Counts of produced sequences of a given length

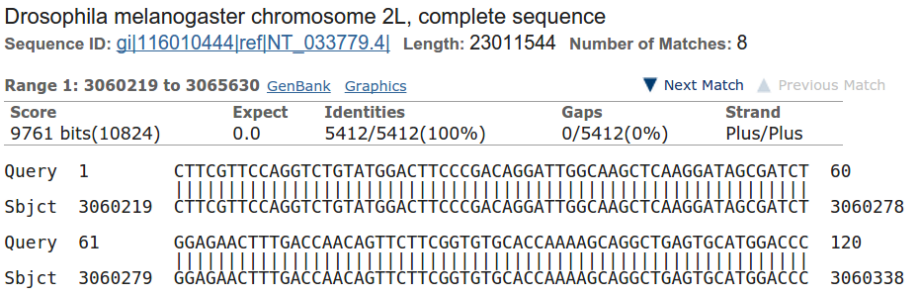


Figure 5: Example output of BLAST for a sequence query

4 Future Work

In the future, we would like to test our sequence assembly method on the Drosophila Melanogaster 2L data with errors. To do this, we must first do error detection on the data using k-mer counting and abundance filtering. We will see how effective these error detection methods are when working with real sequencing data.

We would also like to continue working to improve our method for sequence assembly. We will investigate the possible causes for the erroneous sequence our assembler produced from the clean 2L data. We also plan to add support to our assembler for working with both forward and reverse-aligned reads.

Given the high memory usage ($\sim$4GB) of the program when only sequencing 10% ($\sim$1.3 million reads) of the GSM516588 dataset which contains $\sim$13 million reads, if we want to test our assembler on the complete dataset in the future it will be necessary to port our assembler program to a more efficient programming language, like C++.

5 Conclusion

In this work, we showed an effective method for doing sequence assembly on DNA sequencing data, using the k-mer graph method. We demonstrated that it worked for assembling sequences using clean reads from the 2L chromosome

of the Drosophila Melanogaster reference genome. We also showed that, in the simple case of simulated data, *k*-mer counting and abundance filtering is an effective method for error detection in *de-novo* sequence assembly.

References

[1] By Suspencewl (Own work) [CC0], via Wikimedia Commons: http://upload.wikimedia.org/wikipedia/commons/2/2e/Mapping_Reads.png

[2] Repository of programs developed as part of this investigation (including **assemble.py**, etc.): https://github.com/rlopez93/seq-assembly-2

[3] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[4] Crusoe et al., The khmer software package: enabling efficient sequence analysis. 2014. http://dx.doi.org/10.6084/m9.figshare.979190

[5] screed - short read sequence utils: https://screed.readthedocs.org

[6] Rivera-Hazim, Edwardo S.; Ortiz-Zuazaga, Humberto (2015): A general implementation of Eulerian path. figshare. http://dx.doi.org/10.6084/m9.figshare.1424492

[7] These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure Zhang Q, Pell J, Canino-Koning R, Howe AC, Brown CT. http://dx.doi.org/10.1371/journal.pone.0101271

[8] SeqAn An efficient, generic C++ library for sequence analysis Döring A, Weese D, Rausch T, Reinert K. http://dx.doi.org/10.1186/1471-2105-9-11

[9] Jiang L, Schlesinger F, Davis CA, Zhang Y et al. Synthetic spike-in standards for RNA-seq experiments. Genome Res 2011 Sep;21(9):1543-51.

[10] BLAST: Basic local alignment search tool: http://blast.ncbi.nlm.nih.gov/

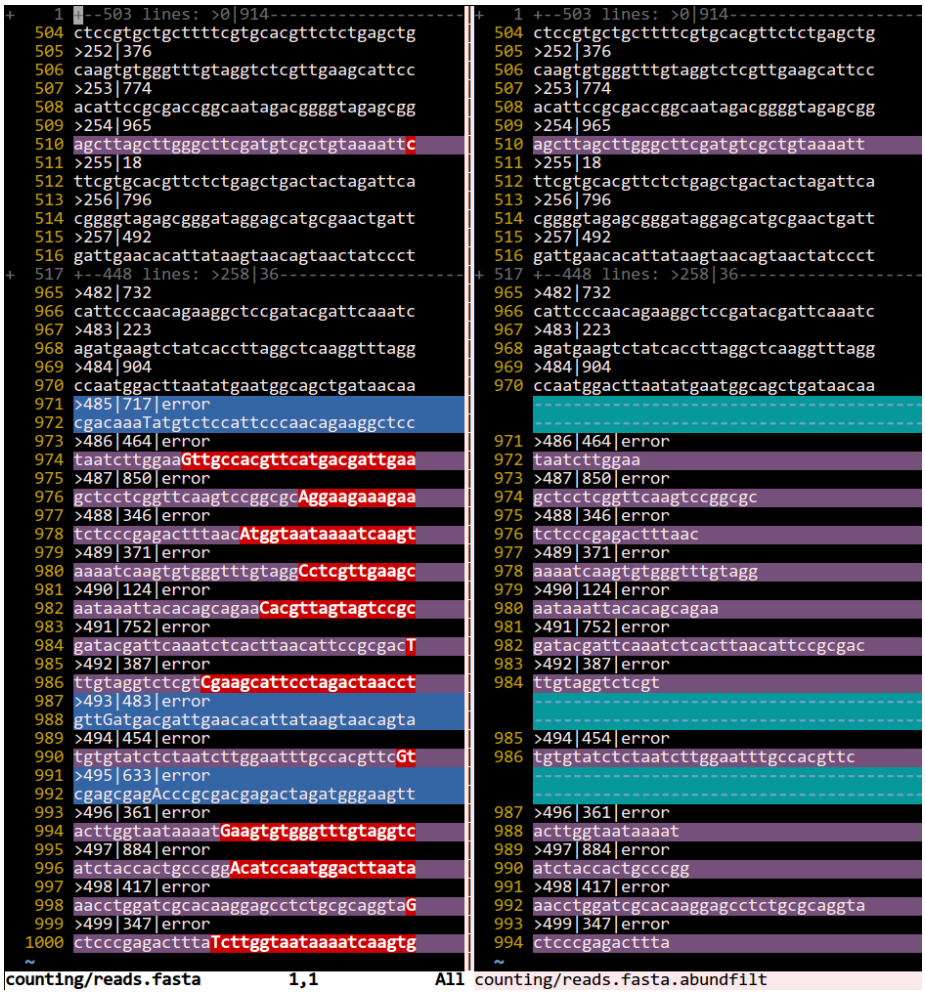


Figure 6: vimdiff of original reads vs. filtered reads

assemble.py

```
from euler import find_eulerian_path
import sys
import screed
import networkx as nx

# input file containing reads
# input may be in FASTA or FASTQ format
infilename = sys.argv[1]

# kmer size to use in building k-mer graph
k = int(sys.argv[2])

# k-mer graph using a NetworkX DiGraph
DG = nx.DiGraph()

# iterate over reads in input file using screed
for record in screed.open(infilename):
    seq = record.sequence  # get current read

    # iterate over all k-mers in seq,
    # and add them to the graph
    for i in xrange(len(seq)-k):
        kmer_a = seq[i:i+k]
        kmer_b = seq[i+1:i+k+1]
        DG.add_edge(kmer_a, kmer_b)

# generator for weakly connected component subgraphs of DG
# each represents a possible contig in the original sequence
wc_subgraphs = nx.weakly_connected_component_subgraphs(DG)

for i, subgraph in enumerate(wc_subgraphs):

    # try block will throw if subgraph
    # does not contain an eulerian path
    try:
        path = find_eulerian_path(subgraph)
        # ...
        # print sequence in path
        # ...

    except nx.NetworkXError:
        # subgraph does not contain an eulerian path
        # so do nothing
        pass
```