Creating Artificial Datasets

Michael Panitz[1] and Mathias Ganter[2]

[1] Department of Computer Science and Engineering, University of Washington, Seattle, USA
[2] Department of Genome Sciences, University of Washington, Seattle, USA

Abstract

DNA sequencing has been a labor-intensive task that has benefited greatly from automation, both hardware based, and software based. Our goal is to develop a tool which can provide challenging input to a software-based sequence assembler, which attempts to reassemble the original DNA sequence given a collection of fragmented reads (or that sequence). In order to accomplish this, it is necessary to examine the DNA sequencing process in detail, and simulate it.

Introduction

This class project was motivated by the needs of a local company that is active within the bioinformatics industry. One of the company's products is a DNA sequence assembly program, and it would be helpful to have a tool that is able to build simulated datasets, to test their assembler. During the course of this project, we have been able to produce a prototype of this tool, and successfully assemble the datasets that it produces, using the company's assembler. This tool currently takes chunks of (previously sequenced) DNA as input, and produces several files that contain data about a collection of "fragments" similar to what one would get from sequencing the given DNA in a wet lab. These files serve as the input to the assembler, which attempts to reassemble these fragments into a consecutive whole.

Our goals are to explore the issues and concepts need to construct such a tool, gain an understanding of the topics involved, and to produce a prototype program that can be expanded into a full version at some point in the future.

Theory

Shotgun Sequencing Simulation

Contact: mpanitz@cascadia.ctc.edu
mganter@u.washington.edu

Our implementation is based on a variation of double-barreled shotgun sequencing. We are not trying to reconstruct our source sequence, but rather, we are trying to create a set of ‘mate pair reads’ given a starting source sequence.

In the lab, the sequencing process is done in a series of steps, which we attempt to simulate (as needed) in our software tool. In the lab, the unknown “initial” sequence is broken random sized subsequences using a physical process (using sonication, or a shearing force). Only subsequences close to a specified length are then filtered out, using a chromatogram. These subsequences are then inserted into one of several cloning vectors, where they are cloned up to levels that are conducive to sequencing. These subsequences, referred to as “fragments” in our program, are also known as inserts, clones, or templates. This is illustrated in Figure 1.

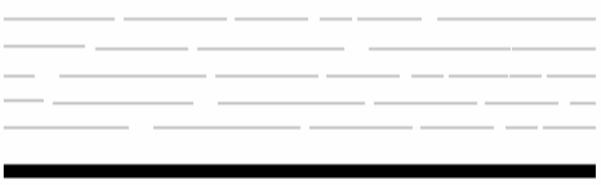

Figure 1: Sequence and sequence fragments
The dark black line is the initial sequence; the light gray lines are the inserts.

We simulate the creation of these inserts by taking our known, initial sequence, and selecting subsequences whose starting position is chosen with uniform randomness. The length is also determined randomly, but is chosen from a normal distribution whose standard deviation is set by the user. Each end of the insert is then read (using the well-known Sanger method) producing two “reads”. Given that the relative position of these two reads are known (within some tolerance), these are usually referred to as “mate pair reads”. Typically, an insert will be about 6,000 bases long, and each of the two reads will be between 750 and 1,000 bases long.

This process creates a set of reads **F** from our sequence **A**:

$$F = \{f_1, f_2, \ldots, f_n\}$$

Essential characteristics of the produced data are a complete coverage of the source sequence (which is usually never achieved in real experiments), same sized fragments, introduction of errors and both strands are taken into account. Note that in an attempt to assure sufficient coverage, it is typical to aim for an average coverage of 10-20 times.

In the lab, this set of reads is then analyzed using an assembler program, which does the work of reassembling as much of the initial sequence as possible.

Sequence Assembly Problem

The assembly problem is now, how to transform the collection of reads over the given nucleotide alphabet into a single reconstructed sequence. This is complicated by mutations amplified by the cloning process, and/or base calling errors from the actual sequencing. Because there are many such solutions and we only want to gain one solution per run, our problem can be considered as a Shortest Common Superstring Problem of the reads with a specified error rate.

This problem consists of three stages:

1. an overlap phase (every read is compared with every other read, and the overlap graph is computed)
2. a layout phase (positions every read in the assembly)
3. a consensus phase (a multi-alignment of all the placed reads is produced to obtain the final sequence)

Note that it may not be possible (given the data) to assemble a single sequence that spans the entire, initial sequence. It is common for assembly programs to instead assemble subsequences called *contigs*. Within each contig, the assembler is reasonably sure of the sequence of bases, but may not be able to position the contig relative to the start of the initial sequence.

To solve the sequence assembly problem, we use the tool *rPhrap*. On the one hand, *rPhrap* is very similar to the operation of *Phrap* that is a program for assembling shotgun DNA sequence data that was developed by Phil Green (University of Washington, Department of Genome Sciences). On the other hand, *rPhrap* has been further developed by the company.

The Quality Function

Most assembler (including both Phrap and rPhrap) require that each base within a read be assigned a *quality score* in order to successfully reassemble consensus contigs. The score is based on the probability of miscalling the specified base. There is strong evidence of a connection between the sequence position, and the likelihood of miscalling the base. For example, the start and end regions have a greater chance to be misread than parts in the middle. Figure 2 illustrates a fairly typical graph of position number (on the x-axis) vs. quality score. Several more reads are given in Appendix A

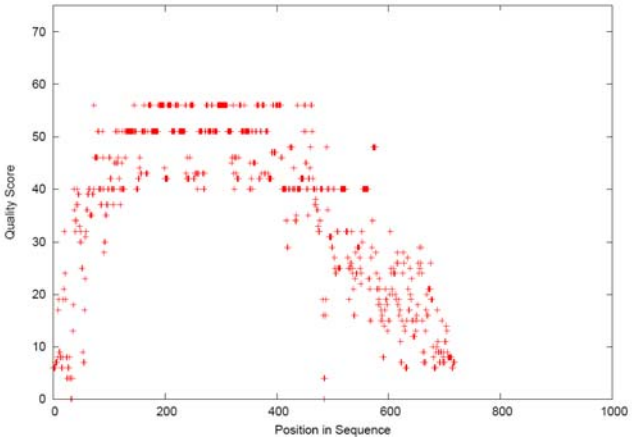

Figure 2: Position in Sequence vs. Quality Score

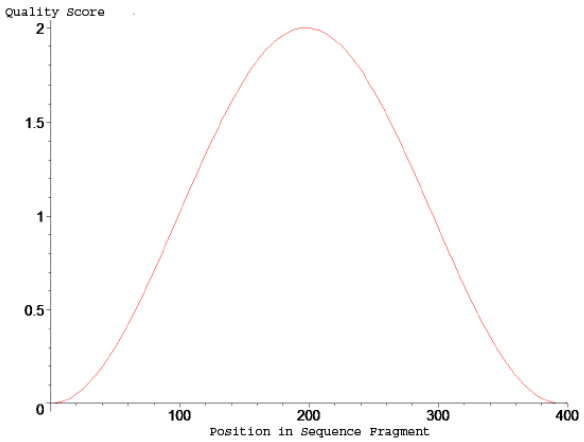

Figure 3: Our Quality Function

In order to simulate this, we needed to come up with a 'quality function' produce reasonable looking values. We came up with a cosine function as an approximate model of quality scores given the data we looked at, which is illustrated in Figure 3.

The quality function assigns to each sequence position of each fragment a quality score. Based on this score, the sequence assembler decides on which nucleotide to choose. The details of our quality function are as follows:

$$score = A \cdot \cos(w \cdot x + q) + z$$

where x is the position in the sequence. The function always ranges from the first residue to the last residue of each sequence segment. The dilation in y direction is always set to a constant value for all segments, this is achieved by choosing a constant value for A.

We chose this function after doing some basic data analysis: we scatter-plotted position number vs. quality score for several individual reads, noting their regular structure. When we attempted to scatter plot numerous different reads onto the same graph, we noted that while individual reads had a regular structure, there was too much variance (especially in the 'middle' region of the read) to do a regression. Lacking further data analysis skills, we decided that the cosine function would offer a sufficiently close approximation for our purposes. Instead of taking this cosine quality function directly into account, we decided to add noise to gain more random data while trying to keep the original plot structure. This effect is also achieved by a cosine function that fluctuates very strongly between two small boundaries.

We decided on the boundaries after making several test runs on various parameters. There is no difference between having a positive, a negative or a positive and negative 'bonus function' as long as the function is symmetrical and keeps its strong fluctuation.

The Implementation

The implementation is written in Java aiming to implement a user-friendly, i.e. readable, code consisting of various classes. Their names refer to their purpose. It is worth mentioning that we started work on our program from scratch and built it all on that. We used *BioJava* which is an open source project that started in 1998 at EBI/Sanger providing us with a library like framework for processing biological data. The fundamental structure of the tool is fairly simple, and is pictured in Figure 4. The tool reads a given sequence from disk using the FASTA file parser provided by the BioJava library. BioJava also provides parsers for a wide range of other file formats, thus allowing us to import data from many different sources. Though not currently implemented, a repetitive element inserter would be useful in order to better simulate DNA that contains substantial numbers of repeated regions. The 'Collector' functionality first generates the set of all inserts, ensuring that the minimum average coverage requirement is met. It then uses the Read Simulator module to actually do the individual reads, which generates a set of Mate Pair Reads, and a set of corresponding quality scores. While not currently enabled, the Mutator module would take each read, and randomly mutate bases within it, thus simulating the occasional mutations that occur when cloning the inserts in the lab. It is also noteworthy to observe that quality scores, which reflect the certainty of correctly reading a base, are separate from mutations, which are a result of flaws in the underlying cloning process. Thus it makes sense to separate out the generation of quality scores from the generation of mutations.

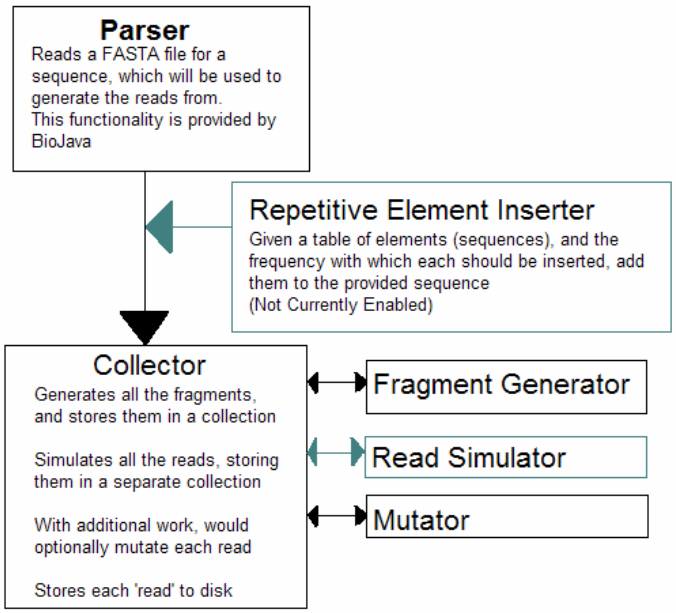

Figure 4: Overview Flowchart of Program

The 'Collector' module then writes two files out to disk: one that contains the reads themselves, and another that contains the quality scores. It would be fairly simple to write additional routines to generate output in other formats, as well.

While this architecture provides a clean, cohesive model to build further features of the tool around, it may require revision in order to scale (see **Areas for Future Work**). However, the tool should be easily able to accommodate new features, such as the ability to generate entirely synthetic sequences. I.e., instead of starting with known, previously sequenced DNA, generating the DNA from scratch, which would allow the authors of an assembler to pick and choose which features they wanted to see in the target DNA.

Discussion

We performed several runs using the human beta globin region on chromosome 11 (gi-no 455025) and the zebrafish DNA sequence from clone XX-187G17 in linkage group 3 (gi-no 24395450). Each run consisted of using our Java tool to produce a file of reads, which was then used by the rPhrap tool to produce a collection of one or more contigs. We then used a normal dotplot viewer (http://arbl.cvmbs.colostate.edu/molkit/dnadot/) and the alignment tool CLUSTAL-W to determine the position of each contig within the initial sequence.

One of the more interesting results that was observed within the zebrafish sequence was that some of the contigs matched the original sequence almost 100% (see Figure 5), whereas other contigs matched closely, but with some significant differences (see Figure 6). This is particular interesting because the Java tool that we developed does not mutate the reads in any way, and furthermore, we assign quality scores in a fairly uniform fashion. This would seem to imply that in the second case, the assembler itself is able to assemble a reasonable-looking contig from the reads that does not exactly match the initial sequence. In particular, the assembler must be getting many regions of the contig correct (thus accounting for the line that's visible in Figure 6), but mixing up the regions in between (thus accounting for the gaps in that line, in the same figure). One possible explanation (or at least a contributing factor) is that while our tool doesn't mutate the initial sequence in any way, assemblers normally expect to see some mutation, and thus must look for less than exact matches between two reads (or subsection of reads). Thus, the assembler may incorrectly 'link' together two reads, ignoring minor inconsistencies between the reads as being mutations, or other forms of noise.

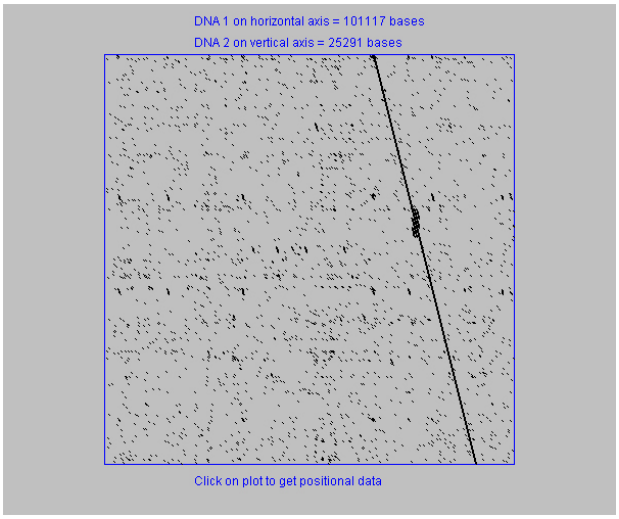

Figure 5: A Nearly Perfect Match Between Contig and Initial Sequence

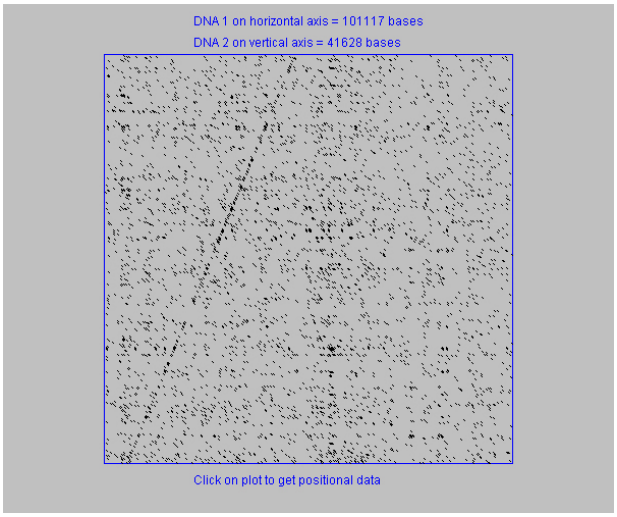

Figure 6: A Less Perfect Match Between A Contig And Initial Sequence

Analyzing methods

We want to start with a brief review about the up to date evaluation, followed by various possibilities for future evaluations. We performed the most basic sequence analyzing task, i.e. a dot plot.

Now let’s consider reasons for choosing the dot plot evaluation method that developed in the early 1980s. A dot plot represents a similarity matrix and it is a visual representation between two sequences. Each axis of a rectangular array represents one of the two sequences to be compared. It is useful for searching out regions of similarity in two sequences and repeats within a single sequence.

The principle used to generate dot plots is straightforward. A matrix comparison of two sequences (or one with itself) is prepared by "sliding" a window of user-defined size (called window size) along both sequences. If the two sequences within that window match with a precision set by the mismatch limit, a dot is placed in the middle of the window signifying a match. Thus, when two sequences share similarity over their entire length a diagonal line will extend from one corner of the dot plot to the diagonally opposite corner. If two sequences only share patches of similarity this will be revealed by diagonal stretches.

Variations in both the size of the sliding window and the stringenthy factor can be used to separate more significant data from less important data.

But, if you want to use a more accurate examination method, one should also take a simple model, like a scoring model, into account to decide on the goodness. There are 3 key issues:

- What kind of algorithms to use to find an alignment (Needleman-Wunsch, Smith-Waterman, FSA-model, HMM, multiple sequence alignments)
- What kind of scoring systems to use to rank alignments
- What kind of statistical methods to use to evaluate the significance of an alignment score

A different approach we though of is using phylogenetic analyzing methods to explore the relationship between various generated sequences and the initial one.

This can be done because we used one initial sequence and built all the other sequences out of this one. One could say that they all diverged from one common ancestor by a simulated process of mutation and selection. This can be interpreted as the relative likelihood that the sequences are related, compared to being unrelated.

Conclusion

At this point, the prototype is mostly complete, in that the initial feature set is implemented, and we have successfully assembled the resulting dataset using the rPhrap assembler. Even at this stage it is clear that the tool is producing interesting sets of reads, as evidenced by the imperfect contigs, and the multiple contigs that the assembler produced (if the reads were trivial to reassemble, one would expect to see a single, (almost) perfect contig result from the assembler.

Areas for future work

The work that has been done so far on quality scores has been done using a reasonable number of reads (~3,600), but all the reads were obtained from a single sequence. It would be good to verify that the underlying physical processes will produce similar results with different sequences, and/or examine data from other sequencing runs to confirm this assumption.

It may be beneficial to better analyze the data for quality scores. For example, there is clearly a low-quality ‘tail’ present in nearly every read (of sufficient length) that was examined by the authors, but the program currently doesn’t model that. Further, there appears to be strong evidence that the quality scores have both a regular overall structure, and substantial variation within certain ranges of base positions.

Generating data sets completely from scratch may be a useful feature as well, and one that will require substantially more research in order to successfully attempt. However, these ‘synthetic’ data sets could both save space, and better control the input to an assembler, thus allowing one to focus one’s attention on a particular issue.

One aspect of synthesizing such data would be to allow the insertion of repetitive elements immediately after generating the ‘raw’ sequence. Further, this technique could be used with sequence that has been obtained from a database, so as to better simulate data that assemblers often encounter in the real world. For example, the program could read in a provided sequence, insert various repetitive sequences randomly throughout the original, and then use that to generate the reads.

On the software engineering side, ‘pipelining’ the read generating process would be beneficial, as the program currently stores all the fragments (and reads) in memory simultaneously. While modern machines are able to perform reasonably well, even for fairly long input sequences (50Kbp -100Kbp), the memory limitations of both the Java Virtual Machine and the underlying physical machine impose upper limits on the number of reads that can be generated. One author believes that it is reasonable to read the entire sequence into memory (that author’s guess is that BioJava will require about 4 bytes per base, plus constant overhead for

each read, and so require ~4MB for roughly 1 million bases), but that the overhead of keeping tens of thousands of fragments and reads in memory will tax a typical machine excessively. Instead, a 'pipeline' design could be used, whereby a single fragment is generated, then mutated, then the reading process is simulated, then the reads are written out to a file, and it's memory released. This would remove any upper bounds on the total number of reads that can be generated, and thus allow the program to generate substantially larger data sets.

An additional feature that would be useful is a tool that could extract information about the precise location of each contig, and the composition of each contig in terms of which reads were used to generate the contig. Even if it's only possible to get the location of read's contribution relative to the start of the contig, it would be an intuitively clear metric with which to evaluate the efficacy of the assembler, and would lend itself to statistical analysis of a more detailed nature.

Another useful feature is one that will mutate the bases prior to writing the fragment/read out to a file. There is actually code partially written to do position-dependent mutation (whereby substitution, insertion, and deletion events can be assigned probabilities of happening based on their position relative to the start of the read), but the authors were unable to find precise, accurate numbers to fill to use for these events, and so disabled the code. It appears that there may be paper(s) available that have studied this topic, and so with a further review of the literature, it should be possible to enable this feature. Alternately, it may be possible to use a third-party, external application, such as the mutate program included in the EMBOSS collection of tools, to do the mutations for us.

Acknowledgments

The authors would like to gratefully acknowledge the help and direction provided by Christie Robertson, who not only furnished the original idea for the project, but helped find resources and materials, including access to a working rPhrap program, during the project.

References

Serafim Batzoglou, David B. Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P. Mesirov, and Eric S. Lander: ARACHNE: A Whole-Genome Shotgun Assembler; Genome Research Vol. 12, Issue 1, 177-189, January 2002

Michael L. Engle and Christian Burks: Artificially generated data sets for testing dna sequence assembly algorithms. *Genomics*, 16:286-288, 1993.

Michael L. Engle, Christian Burks: GenFrag 2.1: new features for more robust fragment assembly benchmarks. Computer Applications in the Biosciences 10(5): 567-568 (1994).

Seto D, Koop BF, Hood L: An experimentally derived data set constructed for testing large-scale DNA sequence assembly algorithms. Genomics. 1993 Mar;15(3):673-6. Related Articles, Links

Gene Myers: Whole-Genome DNA Sequencing, Computing in Science and Engineering, 33-43, May–June, 1999.

Eugene W. (Gene) Myers *et al.*: A Whole-Genome Assembly of *Drosophila*, Science, 287:2196-2204, 24 March 2000.

Soderlund, C., Humphray, S., Dunham, A., and French, L. 2000. Contigs built with fingerprints, markers, and FPC V4.7. Genome Res. 10: 1772–1787.

http://genetics.mgh.harvard.edu/goodman/doc/phrap.html
http://www.biojava.org
http://www.geospiza.com/rphrap/rPhrapInDoc0.html
http://www.phrab.org

Appendix A

These are several more graphs depicting the relationship between position of the base (on the x-axis), and the quality score (on the y axis).