A Dataset Generator for Whole Genome Shotgun Sequencing

Gene Myers
Celera Genomics
Rockville, MD 20850
(e-mail: MyersGW@celera.com)
(tel: +1 240 453-3007)
(fax: +1 240 453-4375).

January 31, 1999

Abstract

Simulated data sets have been found to be useful in developing software systems because (1) they allow one to study the effect of a particular phenomenon in isolation, and (2) one has complete information about the true solution against which to measure the results of the software. In developing a software suite for assembling a whole human genome shotgun data set, we have developed a simulator, *celsim*, that permits one to describe and stochastically generate a target DNA sequence with a variety of repeat structures, to further generate polymorphic variants if desired, and to generate a shotgun data set that might be sampled from the target sequence(s). We have found the tool invaluable and quite powerful, yet the design is extremely simple, employing a special type of stochastic grammar.

Keywords: DNA sequencing, Simulation, Stochastic Grammar.

1 Introduction

The push to sequence the entire human genome is gearing up [1]. We recently proposed a whole genome shotgun approach [2] that is now being undertaken in the setting of a private company [3]. A software system for assembling a 10x data set in 18 months must be capable of incrementally processing 200,000 sequencing reads a day, up to a total of 70 million reads at the end of the project. Tasked with rapidly building such a software system, but with no inputs to work with until production sequencing begins, it was essential for us to develop *celsim*, a simulator capable of generating realistic, repetitive target sequences and shotgun data sets thereof.

In brief, *celsim* consists of three sequential parts. In the first, *sequence generation stage*, one describes a DNA sequence via a stochastic grammar in which elements can be mutated and distributed hierarchically within other elements. It is thus possible to model tandem repetitive arrays, transposon-like repetitive elements, and large-scale duplications. One can further use templates of real DNA sequence if available. In the second and optional *polymorphism stage*, one can then develop a set of polymorphic variations of the target sequence generated in the first stage. This is necessary to model the effect of different donors or DNA sources in some projects. Finally, in the *shotgun stage* one simulates sampling and end-sequencing of any number of insert libraries.

There is to our knowledge only one previously reported simulator of this kind, *Genfrag 2.1* [4, 5]. This simulator permitted one to precisely model the introduction of errors into sampled sequences, but is otherwise significantly superceded by the current work in its ability to model sequences, polymorphisms, and end-sequencing. There are at least three advantages to having such a simulator to work with:

1. One has complete information about the correct solution. This information can be propagated through a software system so that one can build software testers and analyzers that can computationally assess how well a particular phase of the assembly is performing its task. For example, after the overlap phase of our assembler, that attempts to find all pairwise overlaps between reads within a certain stringency, we built an analyzer that reported on the number of true overlaps missed, the number of overlaps induced by each type of repetitive element in the target sequence, and other informative statistics.

2. One can analyze the effect of a particular phenomenon. For example, we can generate DNA sequences that just have micro-satellite repeats in them and no others, and then observe the slippage effect of such elements on overlap detection. As another example, we can generate a DNA sequence that has just ALU-like elements in it and then observe the effect on the layout phase that attempts to build contigs out of individual overlaps.

3. One has available very large data sets that can be rapidly regenerated from small descriptions. For example, we can generate a 10x data set of a synthetic human-like genome of 1/10th scale or 300Mbp in 30 minutes. The description of the data set occupies 1Kb but the data set occupies over 3Gb of disk. We thus have available an entire of library of interesting data sets upon which we can perform timing, optimization, and analysis experiments without the burden of having to securely store them for prolonged periods.

The one caveat with using simulated data is that it only tests for what is simulated. The question arises as to whether *celsim* is sufficiently realistic in that it models all phenomenon that substantially affect the computation. At this time, approximately 6% of the human genome has been sequenced and *C. Elegans*, *E. Coli*, and *S. Cervisae* have been sequenced. Moreover, much work has been done on the analysis of various types of repetitive sequences [6, 7, 8, 9]. It is thus fair to say that much is known about what to model. In addition, we also have available large data sets of real sequencing reads collected from real sequences, and we have also used just the shotgun phase of *celsim* to generate synthetic shotgun data sets of real sequences, e.g., all of *C. Elegans*. To first approximation, the statistics our analyzers produce on these data sets correspond directly with those produced on our synthetic data sets for the organism, indicating that our modeling of the genomes and the sequencing process is, to first approximation, valid.

The remainder of the paper is devoted to a description of the design of *celsim*, primarily from a grammatical point of view, interspersed with examples of its use. We apologize for the simple fixed format syntax of the language, hopefully the reader will appreciate that development time was a key issue for us. What we think interesting here is the design and semantic principles and not the syntax. With just a handful of concepts we can describe and create many interesting and useful data sets. We will conclude with some other applications to which one might put *celsim* to use.

2 Program Interface and Top Level Description

Celsim is written as a UNIX command line tool that may be called with several options. The single mandatory argument is the name of the file containing the specification, and the resulting list of simulated sequence reads is sent to the standard output. There are options to . . .

- . . . specify a specific seed value for the random number generator. Normally *celsim* uses the program process id as the seed to give a different output on every invocation, so this option is essential in the frequent case where one wants to consistently produce the same data set. The seed is also always output by *celsim* so one always knows how to regenerate a given data set.

- . . . produce a FASTA format file of just the read sequences. Normally *celsim* also outputs comment lines that give a detailed description of the instantiation of the grammar describing the target sequence, the polymorphisms introduced into copies of the target, and the source and errors introduced into every read.

- . . . to request the output of the DNA sequence generated, and another to output the instantiations of each sub-element (e.g. repeat) flagged with an @-sign in the specification.

A specification file consists of a DNA specification segment beginning with a ".dna" line, followed by zero or more polymorphism segments beginning with a ".poly" line, followed by one or more sampling segments beginning with a ".sample" line. Formally,

$$\begin{array}{rcl} <\!\mathit{Specification}\!> & \rightarrow & <\!\mathit{DNA\_Segment}\!><\!\mathit{Poly\_Segment}\!>^{\star}<\!\mathit{Sample\_Segment}\!>^{+} \\ <\!\mathit{DNA\_Segment}\!> & \rightarrow & \text{“}\texttt{.dna}\text{”}\ <\!\mathit{DNA\_Spec}\!> \\ <\!\mathit{Poly\_Segment}\!> & \rightarrow & \text{“}\texttt{.poly}\text{”}\ <\!\mathit{Weight}\!>^{+}<\!\mathit{Poly\_Spec}\!> \\ <\!\mathit{Sample\_Segment}\!> & \rightarrow & \text{“}\texttt{.sample}\text{”}\ <\!\mathit{Sample\_Spec}\!> \end{array}$$

and Figure 1 gives an example that we will use extensively. The single DNA segment gives what is effectively a stochastic grammar specifying the structure of the target DNA sequence. Each polymorphism segment specifies the form of a set of mutations that will be applied to the target, each polymorphism modeling a haplotype from the DNA of an instance of the species being sequenced. After the ".poly" keyword one may place a series of real numbers.

A haplotype is generated for each number and the number gives the relative proportion with which each haplotype will be sampled. Each sampling segment specifies a collection of reads to be collected from inserts sampled from the pool of haplotypes specified by the polymorphism segments. If there are no polymorphism segments given, then all reads are sampled from the target sequence specified by the DNA segment.

```
.dna
  A = 150;
  B = A A m(.30);
  C ~ 3-7 p(.2,.3,.3);
@ D = C m(.03) n(10,30);
  S = 30,000,000
         B m(.05,.10) f(.1,.1,.01) n(.10)
         D !(500);
.poly .8 .8
  S .0008
  D 1-1 .00012
  D 2-2 .00006
  D 3-3 .00002
  D 500-1000 .00005
  X 1000-2000 .00005
```

```
.poly .4
  S .001
  D 1-2 .0005
.sample
  48,000
  400 600 .5
  .01 .02
  .33 .33
  .3 1800 2200 .005
.sample
  12,000
  400 600 .5
  .01 .03
  .33 .33
  .4 9000 11000 .015
```

Figure 1: A complete example of a *celsim* specification.

To illustrate consider Figure 1. The DNA segment creates a target sequence that is 30Mbp long. The first polymorphism segment creates two mutated instances, say $H1$ and $H2$, of the source strand, weighted .8 each, and the second polymorphism segment creates another haplotype, $H3$, weighted .4. The first sample segment then generates 48,000 end-reads from inserts of average length 2Kbp, where $.8/2.0 = 40\%$ of the inserts are sampled from $H1$, 40% from $H2$, and 20% from $H3$. Similarly the second sample segment then generates 12,000 end-reads from inserts of average length 10Kbp with inserts sampled from $H1$, $H2$, and $H3$ in the same proportions.

3 Specifying DNA sequences

A *celsim* sequence specification consists of a series of context free production rules where each non-terminal in a right hand side may be qualified by a collection of postfix stochastic operators that orient, mutate, fracture, and replicate the given item. In linguistic terms:

<DNA_Spec> → *<Rule>*$^+$
<Rule> → *<Name>* ("="|"~") *<Container>*? *<Element>** ";"
<Container> → *<File_Name>* | *<String_Constant>* | *<Length><Composition>*?
<Element> → *<Name><Orient>*?*<Mutate>*?*<Fracture>*?*<Repeat>*?*<Regenerate>*?

For simplicity the names of non-terminals are just single upper case letters, limiting one to 26 names. The grammar cannot be recursive, all names referred to must have been previously defined. The string produced in response to the last production is assumed to be the desired target DNA sequence.

There are two types of rules depending on whether the first right hand side item is a container or an element. A container is either:

- <*File_Name*>: a fixed sequence to be imported from a FASTA-formated file, e.g. `A = "</usr/joe/data/ALU.FASTA"`.
- <*String_Constant*>: a manually specified string constant, e.g. `A = "aaaaaaaa"`.
- <*Length*><*Composition*>: a string of *Length* bases where each is randomly chosen with probabilities according to *Composition*, e.g. `A = 150` and `C = 3-7 p(.2,.3,.3)` (as in Figure 1).

In the last example, the length of C is chosen uniformly between 3 and 7, and bases are chosen so that A is generated with probability .2, C and G with probability .3, and T with the remaining probability of .2, giving a GC-rich string. While we could have easily generalized to second- and higher-order Markov models for generating such strings, we did not deem the effect to be significant for our purposes.

A rule that consists of just a container element assigns the sequence of the container to the name on the left hand side. If the container is followed by some number of qualified elements then all the instances of those elements generated by the qualifiers are inserted at random, non-overlapping positions within the container. If the container is specified by length and composition, then the total length of the sequence generated, including the contained elements, equals the specified length. Otherwise the elements are inserted and increase the length of the result. Rules that do not begin with a container, e.g. "`C = A B A;`", concatenate all generated elements in order and assigns the result to the name on the left hand side.

Each element reference consists of a name, referring to a previously defined sequence, followed by an optional list of postfix stochastic operators whose simple syntax consists of a letter followed in parenthesis by a number and/or interval. For example, the phrase "`A o(.8) m(.1,.3) n(6)`", specifies that 6 copies of sequence A are to generated, each mutated with between 10 and 30% point mutations, and occurring in the reverse complement orientation 20% of the time. The point mutations are single base insertions, deletions, and substitutions chosen with equal frequency. The *celsim* mutation operator always applies exactly the requested number of mutations to the element, probabilistically rounding up or down when required so that the average over a large number of mutations would converge on the exact mutation percentage. For example, the rule for D in Figure 1 specifies that it is to consist of between 10 and 30 copies of C each mutated at 3%. Suppose C is of length 5. Consistently rounding down would make D a tandem array of perfect copies and rounding up would give D an overall error rate of 20%. By introducing an error into each copy with probability $5 \times .03 = .15$, *celsim* produces a micro-satellite that is a 3% perturbation of a perfect one. Another fine point, is that one is permitted to specify the number of copies of an element to be inserted into a container as a fraction of the container's size. For example, in Figure 1, 10% of the 30Mbp target genome will consists of the Alu-like B elements.

We also think it essential to model the generation of substrings of given elements. For example, in human DNA many Alu's and LINE's are only partial copies of the full repetitive element. The fracture operator permits one to indicate the percentage of prefix segments, suffix segments, and substring segments of the given element that should be produced. For example for the Alu-like element B in the target sequence S of Figure 1, 10% of the time a prefix or suffix of the copy is inserted, and 1% of the time an interior substring is inserted.

The regeneration qualifier, "!", was introduced to allow a given rule to serve not just to produce one sequence, but to be a template from which many sequences with the same structure can be generated. But in order to hold some parts of the template immutable, we further introduced ~-definitions in which the =-sign is replaced with a tilde, and the interpretation is that the rule is to never be regenerated even if elements referring to it are being regenerated. Consider the three examples below:

```
A = 10;                 A = 10;                 A ~ 10;
B = A m(.05) n(4,8);    B = A m(.05) n(4,8);    B = A m(.05) n(4,8);
S = 10000 B n(5);       S = 10000 B !(5);       S = 10000 B !(5);
```

In the example at left 5 identical copies of a micro-satellite B are inserted into S's container, where B is 4 to 8 tandem copies of A mutated by 5% between copies. In the sample at center, 5 different generations of micro-satellite B are inserted. Each regeneration potentially involves a different number of copies of A, a different sequence for A (since it is also regenerated), and a different set of mutations to each copy. In the example at right, A is ~-defined, so the micro-satellite unit A is the same in all 5 generations, but each copy involves potentially different mutations and a different number of copies of A. In general, these two mechanisms give one complete control over the evaluation of non-terminals in the underlying context free grammar.

4 Specifying Haplotypes

A polymorphism segment consists simply of a series of lines each specifying either a point substitution operation, a deletion operation, or a translocation operation. The syntax is simply:

$$\begin{array}{rcl} <\textit{Poly\_Spec}> & \rightarrow & <\textit{Operator}>^{+} \\ <\textit{Operator}> & \rightarrow & \text{“}\mathtt{S}\text{”} <\textit{Fraction}> \\ & & \text{“}\mathtt{D}\text{”} <\textit{Min\_Length}> \text{“}\mathtt{-}\text{”} <\textit{Max\_Length}><\textit{Fraction}> \\ & & \text{“}\mathtt{X}\text{”} <\textit{Min\_Length}> \text{“}\mathtt{-}\text{”} <\textit{Max\_Length}><\textit{Fraction}> \end{array}$$

The S-operator specifies that the given *Fraction* of the target DNA sequence is to be subjected to point substitutions. The locations of the substitutions are chosen with uniform probability across the target sequence. The D-operator specifies that the given *Fraction* of the target is to be deleted in blocks whose sizes are chosen uniformly from the interval $[Min\_Length, Max\_Length]$, and from non-overlapping locations chosen uniformly across the target. Finally, the X-operator specifies that the given *Fraction* of the target is to be translocated in blocks whose sizes are chosen uniformly from the interval

[*Min_Length, Max_Length*]. Both the source and destination coordinates for a translocation are chosen uniformly across the target.

Nate that there is no operation for inserting sequence. The rationale behind this is that generally the target has a rich repeat structure that would be destroyed by inserting random sequence within it. Moreover, it isn't necessary as deleted sequence in one haplotype looks like inserted sequence from the point of view of another haplotype. Another subtle point is that all deletion and translocation operations are performed first, and thereafter substitutions are applied to the entire potentially shorter sequence at the specified rate. Thus, while deletion and translocation blocks are guaranteed not to overlap, substitutions do occur within translocated blocks.

The first ".poly" segment of Figure 1, gives what should be a reasonably realistic polymorphism model for human DNA if what has been found to be true for the lipoprotein lipase region is true of the entire genome [9]. The specification introduces about .1% SNPs of which 80% are substitutions, 12% are 1-base deletions, 6% are 2-base deletions, and 2% are 3-base deletions. In addition, .05% of the genome will be deleted/inserted in blocks of size .5Kbp to 1Kbp, and another .05% will be translocated in blocks of size 1-2Kbp.

5 Specifying Shotgun Datasets

The specification of a shotgun dataset is simply a series of 8 numbers followed by an additional 4 numbers if end-sequencing of inserts is desired. Formally:

$$\begin{array}{rcl}
<\textit{Sample\_Spec}> & \rightarrow & <\textit{Read\_Spec}><\textit{Pair\_Spec}>? \\
<\textit{Read\_Spec}> & \rightarrow & <\textit{Num\_Reads}><\textit{Min\_Read\_Len}><\textit{Max\_Read\_Len}><\textit{Forward\_Odds}> \\
 & & <\textit{Beg\_Rate}><\textit{End\_Rate}><\textit{Insert\_Odds}><\textit{Delete\_Odds}> \\
<\textit{Pair\_Spec}> & \rightarrow & <\textit{Fail\_Odds}><\textit{Min\_Insert\_Len}><\textit{Max\_Insert\_Len}><\textit{Chimer\_Odds}>
\end{array}$$

The parameter *Num_Reads* specifies the number of reads to be sampled from target(s). The length of each read is uniformly chosen from the interval [*Min_Read_Len,Max_Read_Len*]. Each read is selected from the forward, as opposed to the reverse strand of the target, with probability *Forward_Odds.* The sequence of each read is subjected to the introduction of single base errors, beginning at the start of a read at a rate of *Beg_Rate* and ramping linearly to finish at the end of a read with a rate of *End_Rate.* For example, for the ramp "`.01 .02`" in Figure 1 causes single differences to be introduced into a read at a 1% rate at the beginning of the sequence, increasing linearly to 2% at the end of the read. The final two parameters of the 8 number series, *Insert_Odds* and *Delete_Odds*, give the percentage of the errors that should be insertions and deletions, respectively. The remaining percentage will be substitutions.

If one wishes to further model "double-barreled" shotgun sequencing where both ends of inserts of some size range are sequenced, then an additional four parameters must be given as follows. First *Fail_Odds* specifies the failure rate of read reactions. For example, in Figure 1 this is .3 implying that 30% of all reads will *not* be paired because the read at the other end failed. Then one gives the range, [*Min_Insert_Len,Max_Insert_Len*] from which insert lengths are uniformly selected. In Figure 1, the first sample involves end-reads from inserts that are $2Kbp \pm 10\%$, and the second specifies inserts that are $10Kbp \pm 10\%$. The final fraction, *Chimer_Odds*, specifies the odds with which an insert is chimeric, implying that the two end reads are completely unrelated in terms of their locations within the genome. For our running example, this is set to 1%.

While the sample specification is quite elementary, we find that it is more than sufficient for the purposes of evaluating whole genome shotgun sequencing. Adding features, such as, normally distributed read lengths, or more sophisticated models of sequencing error, while having some second order impact on certain statistics one might collect, will have little bearing on the solvability of the central problem. In fact, a flatter uniform sampling distribution and an absence of specific information about the distribution of errors makes the problem harder, not easier. Our sense is that a design that operates well on such a data set will operate even more accurately given greater information. For example, we ultimately will use quality values associated with reads to further discriminate true from repeat-induced overlaps, but such additional information will only make the job easier.

6 Discussion

From an implementation perspective, *celsim* is a collection of *awk* and *perl* scripts that operate three separate UNIX filters: one for generating DNA sequences, another for producing a single polymorphism, and another for performing a read sampling. There is thus the possibility of recombining these processing elements in different ways if required. In one incarnation, *celsim* outputs a single FASTA file of the sequence reads with the small addition of a large number of comment lines beginning with a "#". These comment lines contain complete trace information on how the DNA sequence was generated, where the various elements in the specification were instantiated, where polymorphisms were induced, where sequence reads came from, and where errors were introduced within them. In a second version, tailored to our environment, *celsim* produces two files, one containing the sequence reads and any audit information about them, and another containing all the generation information otherwise placed in the comment fields of the first version.

We have been using *celsim* for the last several months in the development of an incremental shotgun assembler for whole genome sequencing. We use it both to generate synthetic DNA sequences and synthetic shotgun data sets thereof, as well as to generate synthetic shotgun data sets of existing sequences, e.g. all 75Mbp of *C. Elegans.* This later data set is easily produced with a DNA specification consisting of a single rule whose right hand side is a file name container referring to a FASTA file containing the sequence of the organism. Apart from the benefits described at the outset of this paper, we've also found these data sets to be an excellent software testing vehicle as we can mechanically test output for correctness. Moreover, by varying the amounts of sequence, or the fidelity of repeats, or the level of sequencing error, we have been able to study the robustness, sensitivity, and efficiency of our codes in response to such parameters. Indeed, we are currently testing our codes on synthetic sequences that have repeat complexities beyond those we ever expect to see in human DNA, but may see in plant species.

Apart from its direct use in testing shotgun assembly algorithms, *celsim* can be co-opted to other uses. For example, I have used it in developing and testing algorithms for finding micro-satellites. Along these lines one could also use it to test any repeat finding method. One can also generate two polymorphisms of a given sequence and test a large-scale sequence comparison algorithm. Another possibility is to sample fragments from a DNA sequence of the same size as the sequence, providing a natural input for a DNA sequence multi-alignment program. Yet another use, would be to first sample large BAC sized fragments from a synthetic or imported sequence and then shotgun sample the BACs. This would require a modest reconfiguring the component parts of *celsim*, but would permit the modeling of the sequence tagged connector sequencing and the low-pass shotgun sequencing protocol recently announced by the NIH.

Acknowledgements

The author wishes to thank his colleagues Granger Sutton and Saul Kravitz for help in designing repeat models and for assistance in the development of the software.

References

[1] E. Marshall and E. Pennisi. NIH Launches the Final Push To Sequence the Genome. *Science* 272 (1996), 188-189

[2] J. Weber and G. Myers. Human Whole Genome Shotgun Sequencing. *Genome Research* 7 (1997), 401-409

[3] J.C. Venter, M.D. Adams, G.G. Sutton, A.R. Kerlavage, H.O. Smith, and M. Hunkapiller. Shotgun sequencing of the human genome. *Science* 280 (1998), 1540-1542.

[4] M.L. Engle and C. Burks. Artificially generated data sets for testing DNA fragment assembly algorithms. *Genomics* 16 (1993), 286-288.

[5] M.L. Engle and C. Burks. Genfrag 2.1: New freatures for more robust fragment assembly benchmarks. *Computer Applications in the BioSciences* 10 (1994), 567-568.

[6] C.W. Schmid. Alu: Structure, origin, evolution, significance, and function of one-tenth of human DNA. *Progress in Nucleic Acid Research and Molecular Biology* 53 (1996), 283-318.

[7] A. FA. Smit. The origin of intersperse repeats in the human genome. *Current Opinion in Genetics and Development* (1996), 743-748.

[8] E.E. Eichler. Masquerading repeats: Paralogous pitfalls of the human genome. *Genome Research* 8 (1998), 758-762.

[9] A.G. Clark, K.M. Weiss, D.A. Nickerson, S.L. Taylor, A. Buchanan, J. Stengard, V. Salomaa, E. Vartiainen, M. Perola, E. Boerwinkle, and C.F. Sing. Haplotype structure and population genetic inferences from nucleotide sequence variation in human lipoprotein lipase. *American. J. of Human Genetics* 63 (1998), 595-612.