Sequence Assembly

Part 2

Making the best of automated base-calling

- We (assuming we’re the experts) can set the threshold for strength of signal, and continue the base-calling process only so long as the signal stays above the threshold
- Our software can also compare the sequence with the (known) cloning vector sequence, and remove this material from the end(s) of the output

Sequence Assembly: reality check

- A program like FinchTV is useful for illustrating the results of automated base-calling, and for small-scale editing of specific sequences
- Such manual editing is impractical on a large scale, however; most of the time, in actual practice, we rely on the output of automated base-calling software

Sequence Assembly

- Our goal is to produce a consensus sequence, or contig, that accurately represents the original sequence
  - To produce reliable contig, we need to have multiple fragments covering the same region of the sequence several times
  - This is known as a high degree of coverage: each base position of the original sequence appears multiple times in the fragments

Sequence assembly with CAP

- The data file capdata1.txt contains test data for demonstration of sequence assembly, including:
  - multiple overlapping sequences (from both DNA strands)
  - some improper/ambiguous base calls

```plaintext
>1
ATTTTTAGCAGCTGTTACATCTCCACACACAGCTGCTTATCTCAGGGATTTTTTATCT

>2
ATCGCAAGCACATCTGGTCAACAGCTACCTGTTATCTCAGGGATTTTTTATCT

>3
ATGCGTATCTGTTACATCTCCACACACAGCTGCTTATCTCAGGGATTTTTTATCT
```

```
capdata1.txt
```

```
>1
ATTTTTAGCAGCTGTTACATCTCCACACACAGCTGCTTATCTCAGGGATTTTTTATCT

>2
ATCGCAAGCACATCTGGTCAACAGCTACCTGTTATCTCAGGGATTTTTTATCT

>3
ATGCGTATCTGTTACATCTCCACACACAGCTGCTTATCTCAGGGATTTTTTATCT
```
Using CAP

- Point browser to: [http://host9.bioinfo1.jrom-lee-campus.it/cap](http://host9.bioinfo1.jrom-lee-campus.it/cap)
- Paste sequences into window or use the Browse tool to upload sequences from a file (which gives your job a slightly higher priority on the server)
- Parameters may be adjusted (but defaults are usually OK)

CAP results

The results segment shown illustrates both overlapping and non-overlapping portions of segments 2, 3 and 4

Sequence Assembly from Scratch

- Now we’ll delve into the logic behind sequence assembly programs
- In order to simplify the task, we’ll use some basic (and unrealistic) assumptions:
  - we have error-free data
  - there are no large areas of repeats in the data
- The task can be broken down into three major sub-tasks, listed on the next slide

Steps in sequence assembly

1. Obtain sequence fragments
   - series of DNA fragments from an original sequence
   - we’ll generate our own (artificially)
2. Assemble fragments into contigs
   - string fragments together
   - overlap “prefix” of one with “suffix” of another
3. Validate contig
   - use coverage to determine assembly accuracy, help identify misassembled regions

CAP results

- Top portion gives brief summary of data, followed by list of assembled fragments
- The test data file was set up so that the fragments were in their correct order (from the original sequence) – CAP results reflect this

CAP results

- At the bottom of the page, we see the complete set of contigs assembled by CAP and have the option to select a contig and run a BLAST search

You can select a contig and perform a BLAST search on our local server
Step 1: obtain fragments

- We will need test data for our assembly program
- In order to evaluate program correctness, we will need to know correct assembly of test sequence in advance (although, in general, we wouldn't know this)
- The process of generating test fragments should match, as nearly as possible, how the actual sequencing process would produce data

The real world: how it's done

- Producing fragments using shotgun sequencing entails:
  - using restriction enzymes or PCR with random primers
  - cloning fragments, then sequencing from both ends of cloning vector

Algorithmic approach

- From an initial input string (representing source sequence), generate a set of substrings with:
  - random starting positions
  - random lengths (which fall within defined boundaries to simulate sequencing process limitations)

Fragment generation process leaves us with a problem

- We don't know exactly where (which part of the original sequence) each fragment came from
- Solution: make sure we create enough overlapping fragments to cover whole genome
  - Shotgun sequencing usually tries to achieve 8-fold coverage: every base in the original sequence appears in at least 8 different fragments

Fragment generation algorithm

- Read in original sequence (in FASTA form)
- Obtain parameters from user:
  - minimum fragment length
  - maximum fragment length
  - coverage target ("fold" value)

```latex
\begin{verbatim}
//open(INFILE, "\$5\$in.txt")
//print "error opening input file \$5\$n";
//exit;

my $seq;
while ($seq = <INFILE>) {
    chomp $seq;
    $seq = $seq . " $seq;
}
print "Enter the minimum fragment size: ";
$minfrag = <STDIN>;
chomp $minfrag;
print "Enter the maximum fragment size: ";
$maxfrag = <STDIN>;
chomp $maxfrag;
print "Enter the fold coverage: ";
$fold = <STDIN>;
chomp $fold;

#set the random number generator
random(\$5);

\end{verbatim}
```

- Set up conditions to generate random fragments according to given parameters and store results in text file

```latex
\begin{verbatim}
//open(OUTFILE, "\$5\$out.txt")
//print "error opening output file \$5\$n";
//exit;

\$seqlen = length($seq);

#initialize coverage array so it contains enough slots
#(needed when we check fold)
$@coverage;
for ($i=0; $i<\$seqlen; $i++)
$coverage[\$i] = 0;

#seed the random number generator
random(\$5);

\end{verbatim}
```
Programming and random numbers

- The program fragment on the previous slide contained the following code:
  
  ```c
  #seed therandom number generator
  srand(time($$));
  ```
  
- As the comment states, this code provides a seed value for a random number generator.
- Gee, thanks for clearing that up, right?

Programming and random numbers

- A random number generator is a function that produces, when called, an apparently random number; in Perl, the number produced by the function `rand(x)` will return a real number between 0 and x-1.
- The numbers produced are not actually random, but they appear to be so, especially if you use a seeding function first, as was shown a couple of slides ago.

Fragment generation algorithm

- Generate the random fragments and store them in a text file

```perl
#randomly generate fragments
$fr = 1;
$done = 0;
while ($done==0){
  #get a random length for the fragment
  if (int(rand(2)) returns an integer number >= 0 and < $x
     $length = int(rand($minflag-$minflag+1));
     $length = $maxflag + $length;
  
  #get a random starting position in the sequence
  $start = int(rand($sequence-$length+1));
  
  #get the fragment
  $fragment = substr($seq, $start, $length);
  print OUTFILE "$fragment$n";
  $fr++;
}
```

Fragment generation algorithm

- Continuing loop from previous slide ...
- After fragments are generated, send coverage information to output file

```perl
#update coverage values for the fragments
for ($i=$start; $i<=$start+$length; $i++){
  $coverage[$i]++;
  $done = 1;
}
for ($i=0; $i<=$coverage & & $done==1; $i++){
  if ($coverage[$i] < $threshold)
    $done = 0;
  }
}
```

4/13/2010
Step 2: find largest overlap

1. Read in 2 sequences, set up output file
2. Determine length of maximum overlap between them (length of shorter sequence minus 1)

Algorithm in action

First sequence: CTGACT (length = 6)
Second sequence: ACTCCCTGGA (length = 9)

We line up the sequences like so:

<table>
<thead>
<tr>
<th>First sequence</th>
<th>Second sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTGACT</td>
<td>ACTCCCTGGA</td>
</tr>
<tr>
<td>CTGACT</td>
<td>ACTCCCTGGA</td>
</tr>
<tr>
<td>ACTCCCTGGA</td>
<td>ACTCCCTGGA</td>
</tr>
</tbody>
</table>

No match with n=5, so try n=4:

<table>
<thead>
<tr>
<th>First sequence</th>
<th>Second sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTGACT</td>
<td>ACTCCCTGGA</td>
</tr>
<tr>
<td>CTGACT</td>
<td>ACTCCCTGGA</td>
</tr>
</tbody>
</table>

No match with n=4, so try n=3:

<table>
<thead>
<tr>
<th>First sequence</th>
<th>Second sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTGACT</td>
<td>ACTCCCTGGA</td>
</tr>
<tr>
<td>CTGACT</td>
<td>ACTCCCTGGA</td>
</tr>
</tbody>
</table>

When match is found, we can stop the loop and send the results to the output file.

Algorithm for largest overlap

3. Attempt to match $n-1$ prefix characters from one fragment with $n-1$ suffix characters from other fragment

Step 3: validate contig

- We need to answer this question: is the consensus sequence in fact the same as the source sequence?
- Complications:
  - sequencing errors
  - repeats
- Real-world validation is difficult because we don’t start with a “source” sequence with which we can compare contig; not surprisingly, no single computational solution exists that can guarantee accuracy

Computational approaches to error-handling

- Preprocessing: fix errors in data prior to use
  - improve lab techniques used to generate fragments
  - to handle repeats, ensure that fragments are long enough so that fragment prefixes/suffixes don’t consist entirely of repeats

4/13/2010
Computational approaches to error-handling

- Inprocessing: modify algorithm to handle possible errors; solutions include:
  - introduce threshold value to match when determining overlaps
  - for repeated regions, use *mate-pair* reads: keep track of content at both ends of each fragment; check where each mate-pair ended up in final contig

How repeats lead to misassemblies

- Consider the following original sequence:
  \[
  \text{GTCCTAGGAGTCGTTCTGAG}
  \]
  With the following fragments:
  \[
  \text{GCTA CTCGT TAGGA AGTCG CGTT TCGTAG}
  \]
  We could line them up as follows to obtain a contig:
  \[
  \begin{align*}
  \text{GCTA} \\
  \text{CTCGT} \\
  \text{TGGAGA} \\
  \text{TAGGA} \\
  \text{AGTCG} \\
  \text{CGTT} \rightarrow \text{GTCCTAGGAGTCGTT}
  \end{align*}
  \]

Using coverage statistics

- Can determine base position coverage by counting number of fragments that overlap that position
- Regions with low coverage suggest unreliability: region hasn’t been sequenced enough
- Areas of consecutive bases with especially high coverage may indicate misassembly due to repeats

Back to the real world ...

- Without original sequence, we would not know if contig matches – can’t do direct comparison
- We use coverage as a way to find possible misassembly points

Figure 5.10: Graph of coverage values for each base position in a DNA sequence, with consecutive, high values indicating the possible misassembly of a repeated region