Bayesian Evaluation of de novo Genome Assembly

Sergey Nikolenko1,2, Max Alekseyev1,3, Pavel Pevzner1,4

1St. Petersburg Academic University – Nanotechnology Research and Education Centre of the RAS, 194021, St. Petersburg, Khlopina 8, korp. 3
2Steklov Mathematical Institute, 191023, St. Petersburg, Russia, nab. r. Fontanka, 27
3University of South Carolina, 301 Main St., Columbia, SC 29208, USA
4University of California at San Diego, 9500 Gilman Dr., La Jolla, CA 92039, USA

Bayesian formulation for genome assembly

- How do we formulate the algorithmic side of genome assembly?
- What is the problem, exactly?
- Shortest Superstring Problem (SSP); given a set of strings, find the shortest string that contains all of them as substrings.
- A natural simplification, but in the presence of errors it becomes unclear.
- E.g., [Alkan, Sajjadian, Eichler, 2011] criticize assemblers for making the assembled string too short – which is, from the SSP standpoint, exactly what an assembler is supposed to do.
- If we can specify a suitable generative model for genome sequencing, the model can be used to yield maximum likelihood strings.
- We propose a simple and very general generative model for paired reads (see left).
- As a first application, we propose an evaluation method for assembly quality based on the likelihood.

Generative model

Fix a set of (prior) distributions \( (p_s, p_r, p_d, p_l, p_{err}) \) and a number \( N \).
1. Generate an input genome string \( s \) according to distribution \( p_s \) (this corresponds to our prior knowledge about the genomes of a species or this particular genome).
2. Generate \( N \) read start points \( \{i_1, \ldots, i_N\} \) according to distribution \( p_l \).
3. Generate \( N \) insert lengths \( \{d_{i_1}, \ldots, d_{i_N}\} \) according to distribution \( p_d \).
4. Generate \( 2N \) read lengths \( \{l_{i_1}, l_{i_2}, \ldots, l_{i_N}, l_{i_N+1}, \ldots, l_{i_N+N}\} \) according to distribution \( p_r \).
5. \( R = \emptyset \).
6. For \( j \) from 1 to \( N \):
   6.1 take a pair of substrings \( (s[i_j:i_{j+1}], s[i_j + d_{i_j} + d_{i_j + 1}]) \);
   6.2 introduce errors according to distribution \( p_{err} \), getting \( (f_{j1}, f_{j2}) \);
   6.3 \( R := R \cup \{(f_{j1}, f_{j2})\} \).
7. Output \( R \).

Assembly evaluation

- We propose to rank assembly results according to their likelihood with respect to our generative model.
- For each nucleotide, the error distribution \( p_{err} \) is a property of the sequencing process and can usually be derived from quality values. Probabilities of insertion and deletion errors can be inferred from existing datasets.
- Given the probability of error \( p_{err} \) in a nucleotide, the total log-likelihood of a read \( r \) matching at position \( i \) of a genome string \( s \) is
  \[
  \ell(r | i) = \log p(r | i) = \sum_{j=1}^{\mid r \mid} \log p_{err} (r[j] \rightarrow s[i+j-1]) .
  \]
- The next step is to compute the total probability of a read appearing at the dataset generated from the genome string \( s \). Obviously, \( p(r | s) = \sum_{i} p(r | i) p(i) \), and for uniform coverage we get
  \[
  \ell(r | s) = \log p(r | s) = \log \left( \frac{1}{n} \sum_{i} p(r | i) \right) = \log \sum_{i} p(r | i) - \log n,
  \]
  where \( n = |s| - |r| + 1 \).
- The assembled strings can now be compared by the total log-likelihood with respect to the set of available reads \( R \):
  \[
  \ell(R | s) = \sum_{r \in R} \ell(r | s) = \sum_{r \in R} \left( \log \sum_{i} p(r | i) \right) - |R| \log n.
  \]
  The string that maximizes \( \ell(R | s) \) is the winner.

Implementation issues

- Obviously, direct dynamic programming calculation of \( \ell(r | s) \) (find \( \ell(r | i) \), add them up) is hopelessly slow.
- Moreover, indels happen, which slows it down even more.
- Our current solution uses Bowtie as an external program [Langmead et al. 2009].
- For a set of reads \( R \):
  - choose a set of seeds \( \{r_f\} \) of relatively small length, several seeds from every read;
  - apply the Bowtie fast alignment algorithm to find a set of positions \( i \) where \( r_f \) aligns well with the strings \( s \in S \);
  - output \( \log \sum_{i} p(r | i) - \log n \).

Distributions in the model

- For uniform coverage, \( p_c \) can be assumed to be uniform.
- \( p_d \) is normal or logistic with parameters learned from a specific sequencer’s datasets.
- \( p_l \) is also learned from available sequencing projects.
- \( p_{err} \) can be approximated as a Poisson distribution whose parameters are learned for specific sequencing processes, similar to Quake [Kelley, Schatz, Salzberg, 2010].
- The most interesting part of learning this model will be to specify \( p_s \); this requires insights into the structure of all genomes of a species and most probable mutations.
- Genome priors \( p_s \) are exactly the reason why a Bayesian solution will differ from simply providing the shortest “reasonably correct” string: the shortest string will fit our prior biological knowledge badly.
- This last point is exactly what [Alkan, Sajjadian, Eichler, 2011] is about.
- As a crude approximation, we propose to use a distribution on genome lengths; such a distribution can easily be inferred from any database of available assembled genomes for a certain species.

Extension: priors

- The model can also be easily extended to handle prior information concerning the length of the genome being assembled or the structure of the genome. Suppose that there is a prior distribution on the genome strings \( p(s) \), e.g., inferred from a database of genomes of the same species.
- In that case, instead of maximizing \( \ell(R | s) \) we want to maximize \( p(s | R) = \frac{p(R | s) p(s)}{p(R)} \), and we are therefore maximizing
  \[
  \log p(R | s) p(s) = \ell(R | s) + \log p(s) = \sum_{r \in R} \left( \log \sum_{i} p(r | i) \right) - |R| \log n + \log p(s).
  \]

Extension to de Bruijn graphs

- The same approach can be extended to de Bruijn graphs whenever an assembler outputs the graph.
- Instead of a sum over contigs, we sum over graph edges.
- The difference is that now we can match reads to the end of an edge, spilling out to a subsequent edge.