Notes on redesigning the assembler

Glenn Tesler

July 27, 2010

**Uses to anticipate:**

1. Main focus: Bacterial assembly with reads from a single, haplotype, population, and genome size up to ~ 10 Mb
2. larger haploid genome assembly
3. diploid and polyploid genome assembly, including plants and mammalian chromosomes. There are issues such as SNPs and other polymorphisms between the different chromosomal copies.
4. transcriptome assembly
5. metagenomics: it's ok to avoid this
6. Comparative assembly. Traditionally applies to the scaffolding, but reads could be binned according to where they map on a related genome. OK to avoid this.
7. Repeat graph of existing genome, from an assembly. If the input is the actual assembly, it has different characteristics than reads would, e.g., coverage = 1. So we either have to be cognizant of that, or e.g., create perfect reads from the assembly and pretend we are assembling from reads.
8. A-Bruijn graph applications with an arbitrary size alphabet (e.g., synteny blocks and ESTs). We really should optimize for the 4-character alphabet case, so it is ok to avoid this or to do it indirectly (such as by reviving my codewords scheme).

**Input data to consider:**

1. Nucleotide reads, colorspace reads, and current quality value formats.
2. Unpaired reads. Paired reads. Strobe reads.
3. Current and previous popular platforms, and awareness of what to expect in upcoming platforms.

* Sanger reads & quality values: ~500-1000 bases, read lengths vary, very low coverage.
* NGS basespace: 454, Illumina, of various lengths and with various QV formats.
* ABI SOLiD reads, and possibly mixing SOLiD + basespace reads
* PacBio strobe reads: We should anticipate it and allow for strobe reads as a generalization of mate pairs (such as in Hamid’s read numbering scheme currently allowing 8 subreads). But due to the high error rates, we may not be able to use the reads, at least not alone.  
    
  Also, PacBio and David Haussler are developing a new format for describing sequencing machine calls, that would generalize bases / colors + quality values.

1. Previously assembled contigs:

* Used as input mixed with reads, e.g., in order to improve an assembly by adding in more reads: To the extent that we use read coverage as a factor, we should treat previously assembled contigs differently than reads.
* Previously assembled contigs could also be used for other purposes though:
* Our original purpose of using them was to compute the repeat graph of a known genome
* reference guided assembly
* map reads to a reference to bin the reads for separating them for early stages of assembly

1. Other alphabets: E.g., for rearranged synteny blocks and for alternative splicing graphs based on an alphabet of block numbers instead of nucleotides. If we go back to this, we should consider:

* We should optimize the code for 4-letter alphabets, packed into 2 bits, with 4 bases/colors per byte. For other alphabets, use the codeword scheme I had started to develop.

**Algorithmic issues:**

**Which type of graph to use (need to decide):**

1. k-mer de Bruijn graph for a fixed k  
   with vertices = k-mers of nucleotides in 4-letter alphabet  
   edges at early stages are (k+1)-mers, at later stages are longer
2. A-Bruijn graph still based on nucleotides
3. Overlap graph
4. A-Bruijn graph on alphabets with > 4 letters. This has come up before (rearrangements, alternative splicing graphs). However, we should optimize the software for 4 letter alphabets packed 2-bits per base/color, and use something like the codeword scheme I proposed for other alphabets.
5. In whichever graph is used, we need to decide: as the graph is edited, do we associate each vertex/edge to a single specific sequence, or to the multiple sequences that have been merged together / rerouted to produce it. The proposed read modification scheme may allow us to keep each the sequence at each vertex unique.
6. Another trivial matter to decide about is bad characters in the sequence (‘N’). EULER 2001-4 replaces every bad letter by a random letter in ACGT. Current EULER keeps things as characters during error correction so it can try to correct N’s, and also tries to trim segments with N’s off of reads, to minimize the number of N’s that make it through to the assembly phase. Velvet replaces N’s with A’s.

**Directionality issues:**

1. It should allow double stranded basespace assembly; double stranded colorspace assembly; and true single stranded assembly. There is no reason to fake any of these, if we have it in mind from the ground up.
2. Some messy code in the original EULER and current EULER pertains to keeping duals synchronized. Keeping things synchronized between each vertex/edge and the dual is messy in places in both original & current EULER, and is especially bad for self-dual regions.
3. For reads: original EULER numbered input reads 0,1,...,n-1 and their reverse complements were obtained by adding n.  
     
   Current EULER initially numbers reads 0,1,...,n-1, and uses 2\*i for forwards, 2\*i+1 for reverse.  
     
   Hamid's 64-bit read numbering scheme with various bitfields is good. But it should stay encapsulated so that we can change bitfield widths (and even expand it past 64 bits) in the future if necessary.
4. For vertices and edges: original and current EULER allocate data structures for vertices and edges.  
     
   Original EULER has pointers between a vertex and its dual, and between an edge and its dual. This ate a lot of memory.  
     
   Current EULER does not have a direct way to go between a vertex and its dual. If the vertex is not isolated, you go to any edge on it; the edge has a pointer to its dual; and then you go to the corresponding vertex on the edge. If it is isolated, then there is no way to locate its dual. However, the simplification steps should be symmetric, and include deleting all isolated vertices. This gets a bit convoluted.

**Mate pair issues:**

1. We should allow for single reads, mate pairs, and strobe reads.
2. Hamid’s 64-bit read numbering scheme is good. It allows for multiple files, paired reads, and strobe reads with a certain number of segments.
3. It needs to be configurable for directionality issues. Any given library will follow one particular convention.  
   Pair 🡪 🡪 (equivalent to 🡨 🡨 if it’s on the reverse strand)  
   Pair 🡨 🡪  
   Pair 🡪 🡨  
   Strobe reads: Presumably these will be 🡪 🡪 🡪 🡪 🡪 (all in one direction)
4. Original and current EULER allow for multiple paired-read libraries, each with a simple description of the mate pair separation: the separation either falls in a certain interval, or fails to do so. We should certainly allow specifying a simple interval, but we may want to consider allowing specifying a more complicated empirical distribution based on calibration data such as what Hamid produced and the sequencing companies also produce.

**Three methodologies to deal with duals (not necessarily exclusive):**

1. Velvet scheme for vertices: number the "forwards" vertices 0,1,...,n-1 and obtain the duals by adding n. This is similar to how the original EULER numbered reads and duals.  
     
   Since the number of vertices goes down upon converting from the original graph to the branching graph, and may go down again (or occasionally up) upon graph clean up, transformations, etc., there may be a need to consider garbage collection issues and renumbering. Overall, vertex numbering is simpler than what original & current EULER do, and mathematically represents the same thing.
2. Bidirected graph: with independent arrowheads at both ends of each edge  
   >---> >---< <---> <---<  
   Each vertex and edge represent both a sequence and its dual, depending on which way you go through the edge/vertex.  
     
   There may be pathological cases, such as self-dual sequences, where this is not mathematically equivalent to the separate vertex/dual vertex representation.  
     
   The question is, will graph simplification and other steps be easier or harder with a bidirected graph where we only modify each vertex/edge once and don't have to worry about synchronizing with the duals, than with a graph where we have to keep duals synchronized and treat self-dual objects with special care. It is plausible that a bidirected graph could be easier to maintain, so we should study that question.
3. Forget about duals and just do single stranded assembly: Assume that the reads have sufficiently high coverage. Do all assembly steps without regard to duals. Add a new stage late in the process that attempts to pair up each contig with its dual (or identify a contig as self-dual). There may be

* base-calling inconsistencies: Unfortunate but not a huge deal. A consensus stage can iron this out.
* topological inconsistencies: E.g., a single contig in the forwards direction may be two contigs in the reverse direction, or the connectivity may be different, or the lengths may be dramatically different, etc. There may not be a perfect correspondence between contigs and their duals in this methodology (unlike the other methodoligies where we maintain the correspondence at all times).

**Bookkeeping issues:**

1. During various steps of assembly, the question is how to go between reads on the one hand, and vertices or edges on the other hand. Do we store explicit pointers/numbers/sequences in both directions, or is it implicit, or is the association lost?
2. Original de Bruijn graph, no simplification: Each vertex is a k-mer and the k-mers are unique. Reads are easy to map by locating their k-mers.
3. Condensed de Bruijn graph, no simplification: Some k-mers may be in the middle of edges instead of being vertices.
4. As soon as any graph editing is done, the association between sequences found in a read, and the sequences at certain vertices/edges, may be lost or may no longer be unique, so the question is how to deal with that.
5. 2004 EULER: For every edge, store a list of all read intervals. A read interval consists of
   1. read #
   2. offset of start position within read
   3. length taken from read
   4. offset onto edge
6. The offsets and lengths may be approximate due to indels, merging edges that have different lengths, etc. If we keep the concept of read intervals, we need to improve on this.
7. Directionality issues in read intervals: note that each read and its dual are assigned different numbers, and each edge and its dual are different structures.
8. Insufficient bookkeeping was done during graph simplification, so some reads and mate pair paths could not be threaded through the graph.
9. Current EULER: Uses read intervals. Also uses “alternative edges”: when one edge is merged into another, a map of sequence 🡪 location in graph is produced.
10. Proposal: The read modification proposal may allow us to simplify the bookkeeping. It may be possible to maintain a unique location for each k-mer in the graph (at least before the repeat separation stage). Inconsistencies in read intervals may go away because the modified reads forming an edge will have their lengths modified to be consistent with each other.

**Other past missteps not previously listed:**

1. Erosion should be directly tied to the length of a read tip and its coverage, rather than to shaving a fixed number of nucleotides when we encounter a topological situation that has an association with (but is not 100% equivalent to) a bad read end.
2. Parameters for bulge size, whirl size, erosion, etc.: some of this was geared towards the repeat graph from an existing assembly rather than de novo assembly from reads. Ultimately, it would be better to give parameters that describe alignment quality, or the number of mismatches, indels, etc. So instead of a simple bulge being two edges whose lengths L1 + L2 < bulge size (which allows crazy things like small L1 and large L2), it should be in terms of alignment quality (so L1 and L2 are similar in size). There may be multiple parameters. Even if it is messy to describe multiple parameters in a paper, we should still use multiple parameters if it works better, and we can provide configuration files with the parameters for various sequencers.  
     
   Current EULER uses a mix: some things are in terms of bulge or whirl size. Some things take into account the different edge lengths and sequence similarity. Unfortunately, some values are hard-coded in the code, and apparently geared towards the particular 454 platform for which Mark had data at the time.
3. Also it is bad that parameters are fixed globally throughout the whole graph.
4. In EULER 2004 we used a spanning tree and looked for short cycles in the graph. We picked one path through and deleted the other edges. We did not take edge directions into account, which was bad and led to zigzag paths.  
     
   The current EULER can use a spanning arborescence, avoiding zigzag paths. But, the default is to do something different altogether. It looks for particular types of bulges (two parallel directed paths with same start/end) instead. It’s not specifically tied to paths that any single read actually goes through. It merges edges of the parallel paths together, but introduces problems because the edges may have different lengths, and there may be a different number of edges in the two parralel paths, etc. It considers whirls to be directed loops (edge goes from a vertex to itself). Unfortunately bulge removal may introduce whirls.

**New approach: Graph editing by direct modification of reads**

* Past approach: An initial graph was made from the reads. But then we edited the graph (to remove bulges/whirls/etc.), transformed the graph (based on mate pairs), and the reads did not directly give the graph at these stages.
* In the new approach, we will modify reads so that at all times, the graph (de Bruijn, A-Bruijn, or overlap, depending on which we choose) of the modified reads, is the actual current graph.
* For each read, we need to keep track of the following:
* Read ID. Hamid’s 64-bit ID # is suitable. Previously we used the read name in the Fasta header, and then numbered reads in the file consecutively. It would be good to keep the same ID # throughout all stages, from before error correction & trimming, and through all graph editing phases.
* The number of bases trimmed off the start of the read, and the number of bases trimmed off the end of the read. This may be due to an explicit trimming step where we chop off read ends, and it also may be due to “erosion” of isolated read tips.
* The modified sequence in the middle portion of the read.
* We do not need to keep track of the specific decisions that were made on modifying the sequence, since if that info is needed, the modified middle portion of the read can be realigned to the same portion of the original read (from before read modification & error correction).

General assembly phases:

**Pre-processing and error correction:**

* Hamid’s Avicenna error correction method is tailored to fixed-length Illumina reads. It does not account for variable length reads. It doesn’t tolerate indels. It doesn’t use quality values. It deliberately avoids using read multiplicities. It needs to be tested or adapted for other platforms; if it can be tuned to work well on platforms with other characteristics, that’s great, but we should plan for allowing use of other error correction software (the one from the current EULER; Dima’s SAEC for colorspace; SHREC and improvements to it; David Kelley’s; etc.).
* Trimming of bad ends. Note that for each read, we keep track of how much we chopped off of each end.
* Read labelling: if someone skips error correction or uses other error correction software, we still need to relabel the reads according to Hamid’s 64-bit numbering scheme (assuming we use it). When using someone else’s error correction & trimming, the complete info of what were the original reads, how much was trimmed, etc., potentially will be lost.
* For mixed colorspace/basespace assemblies: Convert everything to colorspace early on (could be before or after error correction, but certainly must be before initial simple assembly). Convert back to basespace later.  
    
  For pure basespace or pure colorspace reads, just stay in that space the whole time.

**Initial simple assembly**

* Form the branching graph from the de Bruijn graph (unless we decide to use a more general A-Bruijn graph or an overlap graph).
* Check if the Simpson & Durbin paper on using the FM Index to construct a string graph, is of possible use.

**Read modification / graph simplification**

* We can rename it, but don’t call every step “Error correction.”
* Erosion of read tips with low coverage: Recall that for each read, we store how much is chopped off from each end. To erode a read tip: increase the amount that is chopped off that end of the read, and update the graph.
* Local deformities such as bulges and whirls of various complexities: the idea is to pick a canonical sequence to substitute into all reads in the bulge/whirl to eliminate it, though some bulges/whirls may be topologically complex. Ideally, we should take a “regional” view instead of a strictly “local” view: given a region of the graph with a bulge/whirl network, see if we can replace the whole region at once (rather than one edge segment at a time), by some sequence that goes through it. At a much later stage, we can reconsider the original sequences and multialignment of them.  
    
  The various inconsistencies that arise in the current way of editing the graph, should be avoided. It may be feasible to modify the graph by simply removing the edges/vertices in the region being edited, and then regenerating them from the modified reads.
* Global issues: We may be able to use mate pair paths and strobe paths during graph simplification, rather than just during repeat resolution and scaffolding:  
    
  Determine paths defined by individual reads, as well as paths defined by mate pairs and strobes if they have a unique routing. Add one to a count for each vertex & edge on this path. These counts may help to decide which edges should be kept and which deleted/merged, and to diagnose chimeric mate pairs. (Chimeric mate pairs are not a big problem; it’s sufficient to ignore the association between the two reads.)  
    
  There is a danger in using mate pair / strobe paths too early: before the graph has been sufficiently simplified, there may be a large number of possible paths between the two reads in a pair. So either this should be done after some graph simplification, or it should be restricted to pairs that have a unique path between them with the correct distance & orientation.
* Chimeric read detection: once detected, chimeric reads should be removed from the graph altogether. (This is chimeric reads, not chimeric mate pairs.)

**Repeat separation**

* Currently we have graph transformation based on paths from individual reads and from mate paired reads. This applies when the two reads of a mate pair are both in the same component and define a path with distance and orientations consistent with the mate pair specs.  
    
  It would be straightforward to extend this to strobe reads, at least when two or more of the subreads are in the same graph component and there is a unique path consistent with the strobe specs. E.g., for strobe (r1,r2,r3,r4,r5) (5 subreads) we could treat it as paths defined by pairs (r1,r2), (r2,r3), (r3,r4), (r4,r5), or we could actually define a single very long path going through r1,r2,r3,r4,r5.
* To estimate the number of repeat copies (which is needed for knowing what to separate by equivalent transformations), the original EULER used a graph flow balancing method. Mark reformulated it in terms of the Chinese Postman Problem. We should revisit this, and also should consider whether, with very high coverage reads, the multiplicities can be estimated based on coverage. Paul Medvedev has prior work that may be relevant.
* We should consider also adding a new stage: separation by multialignment. When we collapse repeats together, the true sequences in the genome may be sufficiently different to distinguish them. We should view graph simplification as a method that clusters reads from different copies of the same repeat together. Then take each contig and use traditional (multi)alignments to separate out the reads from distinct repeat instances, and form consensus sequences of the separate repeat instances. Assume haploid bacteria, since diploid and polyploid samples would complicate this determination.
* We could consider using mixed libraries for repeat separation (and for scaffolding), e.g., assemble based on high-quality Illumina reads only, and use low-quality but longer PacBio strobe reads to define paths for repeat separation (when it threads through a single component) or scaffolding (when the two ends of a long read are in different components).  
    
  Note that if both the short and long reads are of sufficiently high quality, we would just use the longer reads in the assembly in the first place, rather than only using them for later repeat separation and scaffolding.

**Consensus:**

* Go back to the original reads (or an option to go to the error-corrected reads) and use a multiple alignment to form consensus sequences. Note that the amounts that were trimmed off (either due to quality trimming, or to erosion) should probably stay deleted. Original quality values can be brought back in at this stage for the multialignment.
* For mixed colorspace/basespace assembly, we could convert back to basespace at this point.

**Scaffolding based on mate pair / strobe reads only:**

* This is for when the two reads in a mate pair map to different connected components of the graph. It also would apply to strobe reads, when subreads are in different components.
* Determine the relative order, orientation, and approximate spacing, between two contigs. If we use an empirical distribution of mate pair distances, we should be able to give a decent estimate of the spacing.

**Scaffolding based on other information:**

* We probably do not want to go down this route. But other information besides paired reads is often used.
* Low-resolution maps (RH maps, fosmids, etc.)
* Comparative assembly, e.g., the 2x cat genome: supercontigs formed by aligning cat contigs to dog assembly, then grouping consecutive contigs together unless there was explicit evidence (such as from the RH map) to break them.
* As previously mentioned: one could use longer reads (e.g., from PacBio) for scaffolding even if we are unable to directly use them for de novo assembly due to the error rate.

**Visualizations:**

* All versions of EULER since the original one have used GraphViz to show the topology of the repeat graph. It’s easy to generate GraphViz files, but it doesn’t scale very well when there are a large number of contigs. GraphViz does not easily lend itself to showing the sequences on each edge. As an additional project, we should consider either how to improve our use of GraphViz; to use other existing visualization tools; or to create a new visualization tool, for the purpose of dealing with the graph topology and the connection of sequences to the topology.
* The SAM converter that Lars made is great. We will need SAM/BAM output. Since the reads will be modified both due to error correction and due to graph simplification, it is essential that we show how the actual original reads (before such modifications) map to the contigs that are output.