EULER 4 GRAPH SIMPLIFICATION PSEUDOCODE

GLENN TESLER

Additions to the September 7 version are marked with a red comment.

1. Overview of assembly phases

We considered this rough outline:

- **Input:** *OriginalReads*
  One or more disk-resident files of reads, such as FASTA files, including their sequences, and mate pair / strobe information.
  There may also be quality values, or soon, the more general machine state codes in development by David Haussler and PacBio.

- Simple read clean-up: may include filtering reads based on "Illumina purity filter" or poor quality values, and trimming read ends based on quality values. May include discarding duplicate reads.

- Error correction: may use local information (quality values) or global information (ℓ-mer spectra for various ℓ, probably $\ell = k + 1$).

- **Output:** *CleanReads*
  Disk-resident files of reads after preliminary clean-up stages and error correction. This must maintain sufficient information to map each read back to its original read, so that we can produce evidence such as SAM files showing how the original reads off the machine map to the contigs we produce after all our correction/corruption/assembly steps.
  If *OriginalReads* consists of FASTA files generated so that all reads have unique identifiers, it would be sufficient for *CleanReads* to use the same identifiers. The user should generate them with unique identifiers, but they might be careless and fail to do this. Alternately, we may use a numbering scheme such as Hamid's, where each processed read is assigned a new identifier encoding the file it came from; the read number within that file; and which subread number (0 or 1 for paired reads, and more numbers for strobe reads).

- Simple de Bruijn assembly of *CleanReads*.

- Graph simplification in a manner that preserves existence of paths (and their approximate lengths) for paths defined by reads, paired reads or strobe reads.
  The goal is that during the simplification, we do not explicitly do bookkeeping of these paths. Rather, we should do operations that will not lose the paths (and will not introduce other paths), and we should do a minimal but sufficient amount of bookkeeping to allow us to later reprocess the reads file and derive the paths.

- **Output:**
  - Simplified graph.

Date: September 20, 2010.

- Contigs based on the nucleotide sequences along branching paths of the graph.

- A map (later denoted *subs*) from the original set of $(k+1)$-mers to how they are remapped in the simplified graph.

Additional stages, not yet fully fleshed out and agreed upon:

- Map reads, paired reads, strobe reads, to paths in the simplified graph.
  Some paths are unique, some are not.

- Untangling the graph / separating repeats: various methods discussed **but not yet agreed upon** include
  - Topological methods: Equivalent transformations based on paired reads, going in order from the library with shortest insert length to the library with longest insert length.

  - Sequence-based methods: Use multiple alignments to separate either the original or the error corrected reads mapping to each contig, in order to split them back apart.

  - Re-assembly with larger values of k, possibly using "mate reads" meaning that each read pair / strobe that corresponds to a unique path through the simplified graph, is replaced by a longer pseudoread with a consensus sequence along that path. Again we would proceed in an order from shortest to longest insert length library.

- Consensus: Have an option to use either *OriginalReads* or *CleanReads*. Use a multialignment of the reads mapping to each contig to determine the sequence to report for it. This is different than the sequence reported after graph simplification because graph simplification may paste edges together in an arbitrary order, not prioritizing based on coverage.

- Re-assembly by other methods: We discussed that the de Bruijn assembly can be used to get a crude reference assembly that correctly bins together overlapping reads, and merges together different repeat instances. Once reads are aligned to this crude reference, using other assembly methods (overlap-layout-consensus) on portions of the graph.
  It is not clear to me if we would really want to do this, or if it would be better to do some kind of repeat separation and consensus steps based on the original reads (or error corrected reads PROVIDED we use a method whose goal is actual error correction rather than "corruption" that makes all similar repeats identical).

- Scaffolding based on mate pair distances (can be generalized to strobe reads). This is when the two pairs of a read map to different connected components of the graph, particularly if the distance is compatible with exiting through a sink in one component and entering through a source in the other component.

- Scaffolding based on additional reads not used in assembly. For example, PacBio reads are likely to have a huge error rate (15% reported at ISMB), but be very long (1500 bases, potentially spread out in a "strobe"). Although we would not want to assemble from reads with such a high error rate, they may be good enough to scaffold contigs formed from higher quality reads.

- Scaffolding based on other information, such as RH maps, fosmids, comparative assembly, etc. Other people do it. We haven't done this in the past.

- Visualizations: SAM files showing how *OriginalReads* and *CleanReads* map to the contigs. GraphViz files showing the repeat graph / assembly graph. The GraphViz files are cumbersome, so it would be great to develop an alternative.

2. Working with double-strandedness New section

2.1. **Overview.** We are primarily interested in genome assembly, where the reads come from both strands, but we are also interested in single-stranded assembly for applications such as alternative splicing. We have discussed three approaches for dealing with double-strandedness:

(1) **Doubled graph**: Explicitly introduce the dual of each read, vertex, and edge, thus doubling how many of each there are. All past versions of EULER did this.

This method uses about twice as much memory and twice as much time in many steps as the other two methods.

(2) **Bidirected graph**: Just have one copy of each read, vertex, and edge. Use special numbering schemes and a bidirected graph to allow everything to be traversed both ways. Velvet does this. This proposal will use it. Mark Chaisson agrees that in retrospect, it would have been better to do EULER-SR this way.

This method uses less memory than the doubled graph while retaining its accuracy.

(3) **Single stranded graph plus duality reconciliation**: Ignore duality during error correction and assembly. Just assemble as though it's single stranded. High coverage is likely to create dual components. At the end, add a duality reconcilation stage that matches up dual components and reconciles them (via consensus or via deleting one or the other).

Pavel advocates for this. I object to this because it will lead to inconsistencies and the duality reconcilation stage will be messy.

The philosophy of introduce simplifications and clean them up later, sometimes is warranted, but it's not warranted here since bidirected graph is a better solution.

2.2. **Addressing schemes.** These issues are tied together with schemes for addressing reads, vertices, and edges, and their duals.

All past versions of EULER numbered the reads and the number of the dual read was computable from the number of the primary read.

For EULER 2001 and 2004, if there are n reads, then primary reads are numbered $0, 1, \ldots, n-1$ and the dual of read i is $i+n$.

For EULER-SR, initially the reads are numbered $i = 0, 1, \ldots, n-1$. Then we change to primary read $2i$, dual read $2i+1$.

But for vertices and edges, EULER 2001, 2004, and EULER-SR used pointers to Vertex and Edge data structures. There is no mathematical relation between the pointer to a given Vertex (or Edge) and its dual. EULER 2001 and 2004 stored pointers to the dual within each Vertex and Edge structure to the dual. This is complete but increases memory requirements. EULER-SR stores dual pointers for edges but not vertices, and has some messy issues involved.

Velvet numbers vertices similar to how EULER 2001 and 2004 number reads: $i = 0, 1, \ldots, n-1$ and the dual is $i+n$.

In general, it seems the best approach is to have numbering schemes (different than memory pointers) for reads, vertices, and edges, such that it is easy to go between primary and dual IDs just from knowing the ID number, without having to examine anything about the object having that ID number. There are many ways to implement this, including:

- Primary ID $0, 1, \ldots, n-1$; dual ID $n, n+1, \ldots, 2n-1$, with i and $i+n$ in correspondence.
- Primary ID $0, 2, 4, \ldots, 2n-2$; dual ID $1, 3, 5, \ldots, 2n-1$, with $2i$ and $2i+1$ in correspondence.
- Primary IDs ≥ 1, dual IDs are their negatives.
- Primary IDs ≥ 0, dual IDs are their 1's complement.

- Use bit fields, with a certain number of bits for an ID code (does not have to be consecutive numbers) and one bit for primary/dual status. The $Hash + BST$ structure has this form.

2.3. **Bidirected graph.** Components C of the bidirected graph that are not self-dual, may be projected to two dual components C_1 and C_2 in the doubled graph representation. C, C_1, and C_2 each have the same number of vertices and same number of edges.

So for the most part, we can still think of it as an ordinary directed graph when explaining and drawing it, although the data structure is slightly different.

TODO: add pictures

Components C that are self-dual only correspond to one component C_1 in the doubled graph representation, and C_1 may have up to twice as many vertices and edges as C. In EULER 2004 and EULER-SR, there are complications when simplifying bulges, whirls, etc. at a self-dual node. In the bidirected graph, there may be similar complications. For example, in bulge simplification, we are assuming all edges of paths P_1 and P_2 are different, but in a self-dual component, some edges might actually be the same (or the same edge traversed in the dual direction). At minimum, if an operation will not work in this circumstance, we should check for the same or dual vertices or edges being involved and then not do the operation. Beyond the minimum, we could adapt the operations to correctly handle this case.

A related complication: we could have a loop $u \to \text{Dual}(u)$ in the regular de Bruijn graph (e.g., $CGCGCG \to GCGCGC$) or in the condensed de Bruijn graph.

3. Global settings

***k*:** The size of a vertex in the de Bruijn graph.

$k+1$ is the size of an edge.

$k-1$ is the size of the overlap between two vertices that are connected into an edge.

Although we may vary k at other stages of assembly, graph simplification will be done with a fixed value of k.

If we decide that varying k is a noble goal, we should seriously consider either a more general A-Bruijn graph setting, or string graphs.

***Double-Stranded*:** TRUE for double-stranded assembly.

FALSE for single-stranded assembly.

Note that the exact definition of duals depends on whether we are in basespace (reverse complement) or colorspace (reverse only, no complement).

***Alphabet*:** Basespace or Colorspace.

Σ: The alphabet.

For Basespace: $\Sigma = \{A, C, G, T\}$.

For Colorspace: $\Sigma = \{0, 1, 2, 3\}$.

Note that we have previously considered applications with other alphabets, including synteny blocks (markers $1, \ldots, n$ and duals $-1, \ldots, -n$), amino acids (IUPAC has a 26 letter alphabet that's single stranded with no duals), etc. For now we will focus on the 4 letter alphabets; in the future we may consider other alphabets as well. For applications with other alphabets, if duals are allowed, they may still be described as "reverse complement" where the definition of *complement* is adapted to the alphabet.

4. Basic definitions

To represent double-strandedness, we will work with bidirected graphs (where each vertex and edge can be traversed one way for the forwards strand, the opposite way for the reverse strand) rather than with doubled graphs (which earlier versions of EULER used; for every read, vertex, edge, we explicitly introduced its dual).

For a sequence s, the dual of s is the reverse complement (in basespace) or the reverse (in colorspace).

Algorithm 1 Dual(s)

```
if Alphabet=BASESPACE then
  return Reverse complement of s
else if Alphabet=COLORSPACE then
  return Reverse of s
end if
```

To achieve a bidirected graph, we represent each pair of dual edges by whichever is smaller in lex order. The lesser edge is called *canonical*. A sequence s is canonical when:

- In double-stranded mode: s is canonical iff $s \leq \text{Dual}(s)$
- In single-stranded mode: all sequences are canonical.

Algorithm 2 Is-Canonical(s)

```
if Double-Stranded = TRUE then
  return s ≤ Dual(s)
else
  return TRUE                    ▷ Single-stranded: all sequences are canonical
end if
```

5. Set of edges

In the full de Bruijn graph (as opposed to the condensed graph), the edges are $(k+1)$-mers.

We will use an implicit representation of a bidirected graph. Instead of explicitly constructing data structures for vertices and edges, we will form the following:

In double-stranded mode:

$$E_0 = \text{Set of all distinct } (k+1)\text{-mers and their duals in } \textit{CleanReads}$$
$$E_0' = \{\, e \in E_0 \ : \ e \text{ is canonical} \,\}$$

In single-stranded mode:

$$E_0 = E_0' = \text{Set of all distinct } (k+1)\text{-mers in } \textit{CleanReads}$$

There are several ways to represent a $(k+1)$-mer, including:

***ASCII*:** As an ASCII string.

***Packed*:** Use 2-bit encoding of Σ to pack the symbols into a word (e.g., 4 symbols per byte, or 32 symbols per 64-bit word). This requires a 4 symbol alphabet, and does not allow for N's, IUPAC ambiguity codes, etc.

***Raw*:** For 4 letter alphabets, this will refer to *Packed* encoding.
For other size alphabets, this may refer to *ASCII* or other encodings.

***ID*:** An ID number assigned to each $(k+1)$-mer in E_0.
The $(k+1)$-mer can be looked up using this number.
The bits of this number generally do not bear any relationship whatsoever to the nucleotides; this must be treated just as a reference to a $(k+1)$-mer.
The ID numbers must be assigned in a way that allows one to easily go between the ID number of an edge and its dual.
Unfortunately we will have to deal with self-dual strings. We leave out details for now, but we will have to address it. In Basespace, when $k+1$ is odd, there are no self-dual $(k+1)$-mers, but when $k+1$ is even, there are. In Colorspace, there are self-dual strings of all sizes regardless of parity, e.g., a single character repeated $k+1$ times.

We discussed several ways to generate and store E_0', each with its own way of generating an ID number. Here is a quick overview. These may be fleshed out more later. We will need to compare all of these for memory usage and speed.

***Sorted-List*:** Form a sorted list of the entries of E_0'. Assign numbers $0, \ldots, N-1$ to its elements in ascending order. The dual of string number i is assigned the number $-1-i$. (Or we can use $i+N$, or other options.)
One simple (but not necessarily efficient) way to form E_0' is to make a sorted list of the lesser of s and Dual(s) for all $(k+1)$-mers s in *CleanReads*. Then eliminate all duplicates from the sorted list.

***Hash + BST*:** Create a structure consisting of a hash table (hash the $(k+1)$-mers to, say, 24-bit keys) plus a binary search tree for all elements that hash to the same value.
Scan along every read in *CleanReads*. For every $(k+1)$-mer in the read, insert the lesser of s or Dual(s) into this structure.
Thus, elements of E_0' are explicitly inserted into this structure, while non-canonical elements of E_0 are not explicitly in the structure, but their duals are.

Assign an ID number to each element of E_0 where the ID number is comprised of three bitfields: a bit field for the hash key; a bit field for the location of the element on the binary tree; and a single bit to indicate if we are traversing this entry of the hash plus BST structure in primary or dual direction.

Note: For both the sorted list and for the hash table, a potential improvement involves a dictionary index based on the first B symbols, where B is likely in the range of 8 to 12.

For the sorted list, this means having an array of where `AAAAAAAA`, `AAAAAAAT`,... each start, and potentially eliminating these initial characters from what is actually stored (class operations would bring them back in).

For the hash table, this means incorporating the first B symbols literally into the hash key, and eliminating them from what is actually stored (and again, class operations would bring them back in).

***FM-Index*:** Create the Bowtie index and the FM-Index from all reads (and possibly from their duals). Assign an ID number encoding the address of a $(k+1)$-mer based on the numbers the FM-Index assigns to it. Details to be worked out.

This may be an improvement on the concept of the dictionary index, but we will need to check memory usage and speed.

Regardless of which approach is chosen, we need the following.

Algorithm 3 ID-To-Raw(ID)

return Raw representation of the sequence with number ID

Algorithm 4 Raw-To-ID(s)

if s exists in E_0 **then**
  return the ID number of s
else
  return FAIL
end if

Algorithm 5 Cyclic-In(s, c)

s can be represented as $s_1, s_2, \ldots, s_{k+1}$
Return the $(k+1)$-mer $c, s_1, s_2, \ldots, s_k$

Algorithm 6 Cyclic-Out(s, c)

s can be represented as $s_1, s_2, \ldots, s_{k+1}$
Return the $(k+1)$-mer $s_2, s_2, \ldots, s_{k+1}, c$

In addition, we provide several functions that work at the level of ID numbers rather than raw encodings:

- Is-Canonical-ID(ID): TRUE if ID-To-Raw(ID) is canonical, FALSE otherwise.
  ID numbers should be implemented so that this is easy to determine from the ID number without having to examine the sequence.

- Dual-ID(ID): Gives the ID number of the dual sequence.

- Cyclic-In-ID(ID, c), Cyclic-Out-ID(ID, c): These are similar to the versions that work on raw sequences, but they indirectly refer to the sequences by their ID number. If they produce a new sequence that is not in E_0, they return FAIL.

ID numbers should be implemented so that Is-Canonical-ID and Dual-ID are easy to determine from the ID number without having to examine the sequence.

We also need a mechanism to map each ID in E_0' to a payload, where we have the capability of dynamically changing the payload associated with each ID.

- In the *Sorted-List* implementation, we may form an array of size N.

- In the *Hash + BST* implementation, each node of the trees may carry a payload.

- For the *FM-Index* implementation, details of carrying a payload need to be worked out. Hamid suggests that we may be able to insert the concatenation of key plus payload into the structure. We need to determine whether that changes the ID number of the key (since the ID number is based on the FM-Index, which may change as we edit the structure).

We only need to map the canonical IDs (in E_0') to a payload; if ID is not canonical, then payload(ID) is deduced from payload Dual-ID(ID) by taking its dual.

6. Elemental graph editing operations

6.1. **Initial graph.** The initial graph G_0 has vertices

$$V_0 = \{\, s_1 \cdots s_k, s_2 \cdots s_{k+1} \ : \ s_1 \cdots s_{k+1} \in E_0 \,\}$$

and edges

$$\{\, (s_1 \cdots s_k, s_2 \cdots s_{k+1}) \ : \ s \in E_0 \,\} \ .$$

Note that E_0 is a set of strings, not a set of ordered pairs of vertices; therefore E_0 is not literally the set of edges of a graph, but we are treating it as the edges of the graph based on the above correspondence.

6.2. **Bookkeeping philosophy.** As we edit the graph, the set of vertices and edges will change.

Past versions of EULER stored information on every edge about which intervals of which reads were mapped to the edge, and then as the graph was edited, the lists of read intervals would be moved around.

Instead of doing that, we will maintain information on how the graph is edited. After all edits are done, we can rescan the *CleanReads* file and remap the paths determined by its reads and mate pairs, one at a time, according to how the edges of its initial path were remapped in the edited graph.

6.3. **Graph operations considered.** We considered different graph editing operations, and also whether editing should be done using local or global information. The editing operations we considered include: (TODO: pictures)

Vertex deletion: Minor changes Delete a vertex that has no incident edges. This can only be done after all edges on it have been deleted or remapped elsewhere. In earlier versions of EULER, we explicitly had to delete vertices. Since we are now working with an implicit graph representation based on E_0, we may not have to explicitly delete vertices.

Edge deletion: This will be used for chopping off read tips, and may be used for removing chimeric edges. New: We also may delete low coverage edges.

Vertex or edge duplication: We may need to do this for equivalent transformations or other methods of separating repeats. However, it will not be needed in graph simplification.

Vertex-pasting: Form a quotient graph by pasting two vertices together. We decided to avoid this operation.

Edge-and-vertex pasting: Form a quotient graph by pasting edge $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ together, as well as pasting vertices u_1 and u_2 together, and v_1 and v_2 together.

EULER-SR uses this operation. We decided to avoid this operation in the new assembler since it can increase vertex degrees by a lot, and since it can create connectivity that did not previously exist.

Edge-only pasting: pasting edge e_1 onto e_2 means that $(k+1)$-mers that initially were represented by e_1, are now considered to be represented by e_2. We do not paste the vertices, however.

We decided to use this operation because it results in a subgraph of the original graph. However, the operation must be applied with great care. Consider a path in the original graph comprised of edges $(e_1, e_2, \ldots, e_m)$. This satisfies the conditions defining a path: the head of e_i equals the tail of e_{i+1}, for $i = 1, \ldots, m-1$.

After one or more edge-only pasting operations, these edges are mapped to $F = (f_1, \ldots, f_m)$. Since we only pasted edges and not vertices, it is possible that the head of f_i is different from the tail of f_{i+1}. We must ensure that F, while not necessarily contiguous in the graph, can be repaired into a true path closely approximated by the edge sequence F.

6.4. **Graph operations to implement.** We decided to use two of these operations: edge deletion, and edge-only pasting.

Restricting to these two operations guarantees that at all times, the edited graph is a subgraph of the original graph (rather than a quotient graph or other type of derived graph).

To facilitate edge deletion, we introduce an ID number DEL for a fake edge that does not exist in the graph. For convenience of implementing the Dual-ID function, we may also introduce an ID number DEL' = Dual-ID(DEL) for its dual.

- It is best to set DEL to a constant that is impossible to have with any data set. However, we may also use a data-dependent ID code: determine a $(k+1)$-mer $D \notin E_0$. Augment E_0 by adding in Canonical(D), and set DEL to its ID number.

- In the *Sorted-List* implementation, we may take DEL to be `INT_MAX` (or the maximum in another integer type if we use a different width integer). Or we may number the edges $1, \ldots, N$, their duals Dual-ID$(i) = -i$, and assign DEL=0.

We will implement these operations by creating a map $subs[ID]$ that maps IDs of edges in E_0 to IDs of edges in E_0, or to DEL or DEL'. This map has the following properties:

- We only store $subs[e]$ for $e \in E_0'$. We do not store it for non-canonical edges. Instead, for each canonical edge e and its dual e', we define $subs[e'] =$ Dual-ID$(subs[e])$.

- At all times during graph editing, the edited graph has edges represented by the $(k+1)$-mers of E_0 whose ID number is unchanged by $subs$.

6.5. **Union-Find implementation.** We will implement edge deletion and edge-only pasting using an adaptation of the "Union-Find" structure. The adaptations we make are as follows:

- We implicitly support dualization of $(k+1)$-mers.

- The graph editing operations that we support will at all times give a subgraph G of the original graph G_0 (rather than a quotient graph or other derived structures). The representative of each edge class must be the unique edge of G it contains. Since we require a specific representative, heuristics for speeding up Union-Find based on arbitrary choice of representatitve (such as the "union by rank heuristic") are not applicable.

- Our "*subs*" map corresponds to the "*parent*" map in textbook descriptions of Union-Find.

Let IDS_0 denote the set of all ID numbers of edges in E_0.
Let IDS_0' denote the set of all ID numbers of edges in E_0'.
As we delete edges from the graph, and as we paste edges to other locations, we will form a subgraph of G_0. We will do this by updating the map

$$\text{subs} : IDS_0' \to IDS_0 \cup \{\text{DEL}, \text{DEL'}\}$$

Initially, there are no substitutions; all edges are represented by themselves:

Algorithm 7 INITIALIZE-EDGES(IDS'_0)

for all $ID \in IDS'_0$ **do**
  $subs[ID] \leftarrow ID$
end for
$subs[\text{DEL}] \leftarrow \text{DEL}$

As we delete edges or remap edges, we form a subgraph of the original graph. To check if an edge is in the subgraph, we do the following:

Algorithm 8 IS-EDGE(ID)

Require: $ID \in IDS_0$
  if Is-Canonical-ID(ID) **then**
    return $ID = subs[ID]$ ▷ Boolean: did the ID change?
  else
    return Dual-ID(ID) = $subs$[Dual-ID(ID)] ▷ Boolean: did the dual ID change?
  end if

Pasting edge e onto edge f means that e, f are currently edges in the edited graph, but after pasting, e will no longer be in the graph; instead, anything mapping to e should instead be remapped to f. This pasting operation corresponds to the Union operation in the Union-Find structure. It has the adaptation that we must use f as the representative rather than allowing e, f, or anything equivalent to either of them.

Algorithm 9 PASTE-EDGE(e, f)

Require: Is-Edge(e) **and** Is-Edge(f)
Require: $e \neq f$ **and** $e \neq$ Dual-ID(f)
  if Is-Canonical-ID(e) **then**
    $subs[e] \leftarrow f$ ▷ Remap edge e to edge f
  else
    $subs$[Dual-ID(e)] $\leftarrow$ Dual-ID(f) ▷ Remap dual of edge e to dual of edge f
  end if

Algorithm 10 DELETE-EDGE(e)

Require: Is-Edge(e)
  if Is-Canonical-ID(e) **then**
    $subs[e] \leftarrow \text{DEL}$ ▷ Remap edge e to "deleted"
  else
    $subs$[Dual-ID(e)] $\leftarrow$ DEL ▷ Remap dual of edge e to "deleted"
  end if

After all graph editing operations are complete, it may be necessary to follow a chain of substitutions to determine how to remap an edge. We implement this as follows, along with the "Union-Find path compression heuristic."

Algorithm 11 Remap(e)

```
if e = DEL or e = DEL' then
  return e                          ▷ This clause may not be needed, since subs[DEL] = DEL
end if

f ← subs[e]
if e = f then
  return e
else if e = Dual-ID(f) then
  return f        ▷ One should avoid mapping an edge to its dual, but we catch it just in case.
else if not Is-Canonical-ID(e) then
  return Dual-ID(Remap(Dual-ID(e)))
else
  subs[e] ← Remap(f)
  return subs[e]
end if
```

After all graph editing is complete, we apply the Union-Find "path compression heuristic" to all canonical edges of the original (pre-edited) graph:

Algorithm 12 REMAP-ALL

```
for all e ∈ IDS_0 do
  Remap(e)
end for
```

Note that it may be more efficient to implement Remap in a non-recursive manner:

Algorithm 13 Remap(e) implemented in two passes without recursion

```
        ▷ First pass: Determine f (root of this edge class) and flip (whether or not to dualize it)
flip ← false
f ← e

while f ≠ DEL and f ≠ DEL' and f ≠ subs[f] and f ≠ Dual-ID(subs[f]) do
  if not Is-Canonical-ID(f) then
    flip ← not flip
    f ← Dual-ID(f)
  end if f ← subs[f]
end while

if flip then
  r ← Dual-ID(f)                                          ▷ Save as return value
else
  r ← f
end if

                                  ▷ Second pass: Modify all entries on chain from e to f.
r_0 ← r
r_1 ← Dual-ID(r)
flip ← false
f ← e

while f ≠ DEL and f ≠ DEL' and f ≠ subs[f] and f ≠ Dual-ID(subs[f]) do
  if not Is-Canonical-ID(f) then
    flip ← not flip
    f ← Dual-ID(f)
  end if
  g ← subs[f]          ▷ Save next entry in chain since we are about to modify subs[f]
  if flip then
    subs[f] ← r_1
  else
    subs[f] ← r_0
  end if
  f ← g                                                   ▷ Advance to next entry in chain
end while

return r
```

6.6. **Coverage** **New section.** In the past we have also kept track of coverage. We can keep track of that here too with an additional map from edges to counts.

We would have to consider, should we modify PASTE-EDGE(e, f) so that the coverage of e is transferred to f.

7. Graph simplification

In the past we had several graph simplification operations:

Low coverage removal: New Delete edges with coverage below a threshold.

Erosion: This serves two purposes: cutting off bad read ends (which can be done at the beginning), and cutting off short paths that jut out after deleting edges from bulges (which must be done after other graph simplification operations).

Bulge removal: Two or more parallel paths are formed due to mismatches or indels. We replace one path by another.

Whirl removal: A short directed cycle is formed in the graph and we remove it. These mainly arise from artifacts in how bulge removal was previously implemented, so if we are careful, we should not have to deal with these any more. In principle, it can also arise from exact tandem repeats where the total length is larger than k and the same k-mer occurs twice; in this case, it is real rather than an artifact of graph simplification. It probably should not be removed. Re-assembly with larger k, or use of another assembly method (such as A-Bruijn graph or string graph or overlap/layout/consensus, etc.) could sometimes handle it, so we should consider whether to make a special case of it for our de Bruijn graph framework.

Zig-zag removal: This was an artifact of the operations allowed in the 2004 EULER. We will do things differently so that this artifact does not occur.

Thorns: TODO

7.1. **Erosion.** There are two different philosophies for how to implement erosion.

The first is based on a fixed threshold, and may be suitable for clipping bad read ends as well as for cleaning up stray edges after bulge removal. The second is based on actual reads, and may be better suited for clipping bad read ends (but is not suitable for cleaning up stray edges after bulge removal).

Note that a path comprised of edges $e_1, \ldots, e_m$ represents $k + m$ nucleotides.

Let T denote the *erosion threshold*; stray paths (one end is a source or sink) contributing fewer than T vertices will be removed. The length includes the nucleotides in the source/sink, and intermediate nucleotides, but not the k nucleotides in the branching vertex on the other end.

Algorithm 14 Erosion based on topology only

for all Vertices v in G that are sources or sinks **do**
  Form the directed path $P = (e_1, e_2, \ldots, e_m)$ emanating from v and terminating at either a branching vertex or another source or sink w.
  if $m < T$ (meaning P has fewer than $k + T$ nucleotides) **then**
    for $i = 1$ **to** m **do**
      DELETE-EDGE(e_i)
    end for
  end if
end for

Algorithm 15 Erosion based on reads

Require: $G = G_0$ ▷ Can only apply to the graph before any other simplification steps have occurred
for all Read r in *CleanReads* **do**
  r corresponds to a path $e_1, \ldots, r_m$ in the original graph G_0.
  if the tail of e_1 or the head of e_m has degree 1 and some intermediate vertex does not have In-Degree(v) = Out-Degree(v) = 1 in G_0 **then**
    for $i = 1$ **to** m **do**
      DELETE-EDGE(e_i)
    end for
  end if
end for

We could also have a coverage count for every edge, and make use of it in deciding whether to erode.

Note that the bulge removal operation outlined below will not usually introduce new vertices of degree 1. (The only exception turns out to be a bulge on a source of out-degree > 1 or a sink of in-degree > 1; intuition says this is unlikely to happen a lot, but we should check.) Therefore it may be sufficient to use only one pass of Erosion, after creating the graph and before any other graph editing operations are done. Erosion based on reads is suitable for this.

However, if we decide to use other bulge removal methods, they may have to be interleaved with erosion passes.

Warning: Note that erosion takes care of obviously bad read ends, but it is possible that a small appendage (say 1 to 3 nucleotides) may survive because it happens to be the same as valid sequence that exists in other reads. If this small appendage leads through a branching vertex in the graph, it may complicate efforts to form mate pair paths later on.

7.2. **Bulge removal.** TODO: This section is messier than the others and needs to be cleaned up.

Point mutations and indels may result in parallel pairs of paths (expressed in vertices) between two vertices v and w:

$$\begin{aligned} P_1 &: v = t_0 \to t_1 \to \ldots \to t_L = w \\ P_2 &: w = u_0 \to u_1 \to \ldots \to u_M = w \end{aligned} \tag{1}$$

These paths share initial vertices ($t_0 = u_0 = v$) and final vertices ($t_L = u_M = w$) but not intermediate vertices, and hence do not share any edges.

A single point mutation results in a bulge with $L = M = k + 1$.

Insertion of y consecutive nucleotides into the read traversed by P_1, or deletion of y consecutive nucleotides from the read traversed by P_2, where $1 \leq y \leq k$, results in $L = k + y$, $M = k$.

Successive point mutations or indels closer than $k + 1$ nucleotides apart will result in larger bulges.

In EULER 2004, the length of a bulge was determined by treating the bulge as an undirected cycle and computing the number of nucleotides in it. A spanning tree was formed. Edges were selectively added back in, provided they did not form undirected cycles below a certain length threshold. Ignoring edge directions led to artifacts in the graph.

In EULER-SR, the bulge is two directed paths with the same start and end, but the bulge size is based on the sum of the two path lengths. This is still bad because we should be prioritizing paths of equal length (substitution errors) and then paths of close lengths (indel errors). Initially, EULER-SR removed bulges using a spanning arborescence instead of a spanning tree. Currently, it is achieved by finding two parallel paths, then merging them together one edge at a time using the edge-and-vertex pasting operation. If the paths have different lengths, this eventually results in whirls (short directed cycles) on the residual edges.

We will employ a different method. We will find pairs of parallel paths P_1, P_2 with the same initial vertices v and same terminal vertices w, where the paths have equal or similar lengths, below a certain threshold. Then we will find a subpath Q_1 of P_1 with no additional incoming/outgoing edges and a subpath Q_2 of P_2, and use edge-only pasting to paste Q_1 to Q_2.

In order to avoid destroying other paths that share edges with the bulge but either enter it after v or exit it before w, we choose Q_1 as follows. See Eq. (3) for notation.

Equations corrected Let

$$\begin{aligned} &i \text{ be maximal in } 0 \leq i < L \text{ with Out-Degree}(t_i) \geq 2 \\ &j \text{ be minimal in } 0 < j \leq L \text{ with In-Degree}(t_j) \geq 2. \end{aligned} \tag{2}$$

Since Out-Degree(t_0) = Out-Degree$(v) \geq 2$ and In-Degree(t_L) = In-Degree$(w) \geq 2$, these are both well-defined.

TODO: Picture

If $i \geq j$ then we cannot safely paste part of P_1 to part of P_2.

If $i < j$ then we may paste the subpath $Q_1 = (t_i, t_{i+1}, \ldots, t_j)$ to the corresponding positions within P_2 using edge-only pasting. However, there are two types of complications:

- If $0 < i$ or $j < L$ (or both), the pasting is called *partial*.
- If $L \neq M$, the pasting is called *mis-sized*.

Pastings that are partial or mis-sized will complicate efforts to remap reads later on, but not make them impossible.

Algorithm 16 PASTE-SUBPATH(P_1, P_2, i, j)

inputs
  $P_1 = (e_1, \ldots, e_L)$ expressed in terms of edges
  $P_2 = (f_1, \ldots, f_M)$ expressed in terms of edges
Require: $i < j$
Require: All intermediate vertices along $e_{i+1}, e_{i+2}, \ldots, e_j$ have in-degree 1 and out-degree 1
if $L = M$ **then**
  for $k = i + 1$ **to** j **do**
    PASTE-EDGE(e_k, f_k)
  end for
else
  for $k = i + 1$ **to** j **do**
    $r = \text{round}(i \cdot M/L)$
    PASTE-EDGE(e_k, f_r)
  end for
end if

We will use at least two parameters to remove bulges:

***B*:** To be considered for bulge removal, two parallel paths must both have length $\leq k + B$.
$B = 0$ effectively disables bulge removal.
$B = 1$ allows for isolated single point mutations.
$B > 1$ allows for more complex situations.

***Expansion*:** To be considered for bulge removal, two parallel paths must have lengths that differ by at most *Expansion*.

Bulge removal is done most efficiently using the condensed graph representation. Details for the implicit graph representation will have to be worked out, and/or the *subs* table and operations should be adapted to the condensed graph.

A *simple bulge* is two parallel paths with the same initial vertex, the same final vertex, and all intermediate vertices are 1-in/1-out. Since the in-degree of a vertex is at most 4 and the out-degree of a vertex is at most 4, there are at most $\binom{4}{2} = 6$ simple bulges emanating outwards (or separately, emanating inwards) to any vertex.

Algorithm 17 Simple-Bulge-Removal(B,*Expansion*)

for all Vertices v with Out-Degree(v) ≥ 2 **do**
  Form the outward condensed paths from v.
  If any pair of them P_1, P_2 each have length $\leq k + B$ nucleotides, and the path lengths differ by $\leq$ *Expansion*, then use PASTE-SUBPATH to paste all of P_1 to P_2.
end for
return Number of edges pasted

A *complex bulge* is two parallel paths with the same initial vertex and the same final vertex, but one or both of the paths has an intermediate vertex with in-degree or out-degree not equal to 1.

New paragraph In terms of the condensed graph: P_1 consists of $r \geq 1$ segments in the condensed graph, and P_2 consists of $s \geq 1$ segments. The bulge is simple if $r = s = 1$ and complex if either $r > 1$ or $s > 1$.

Algorithm 18 Complex-Bulge-Removal(B,*Expansion*)

1: **for all** Vertices v with Out-Degree(v) ≥ 2 **do**
2: Do a bounded depth-first traversal of G to mark all vertices within a distance B nucleotides of v.
3: If some vertex w is marked twice, we have a pair of parallel paths P_1, P_2. Since we are doing a depth-first search, these paths are minimal in the sense that we cannot clip off their initial or final edges and still have parallel paths.
4: **if** $|\text{length}(P_1) - \text{length}(P_2)| \leq$ *Expansion* **then**
5: Compute the subpath of P_1 from i to j as described in the text.
6: **if** $i < j$ **then**
7: PASTE-SUBPATH(P_1, P_2, i, j)
8: **end if**
9: **end if**
10: **end for**
11: **return** Number of edges pasted

Following material is substantially rewritten
We should prioritize bulges to collapse based on their complexity:

- Simple bulges with both paths of equal or close length (in nucleotides) first.
- Complex bulges where P_1 is one condensed segment ($r = 1$) and has identical or similar length (in nucleotides) to P_2. That is, P_1 is one condensed edge, and we are pasting it in full to the whole of P_2.
- Complex bulges where the two parallel paths have equal length, and we are pasting the first condensed edge of P_1 to the first condensed edge of P_2, and these condensed edges have equal length. The dual of this (pasting last condensed edges) is also of equal priority.
- If we allow more elaborate cases, continue to list them in an order that tends to preserve lengths and positions along the path.

Note that EULER-SR only pastes (edge-and-vertex pasting) condensed edges both coming out from the same vertex.

Algorithm 19 BULGE-REMOVAL(B,*Expansion*)

repeat
  progress ← Simple-Bulge-Removal($B, 0$)
  if *progress* > 0 **then**
    Restart loop
  end if

  progress ← Simple-Bulge-Removal(B, *Expansion*)
  if *progress* > 0 **then**
    Restart loop
  end if

  progress ← Complex-Bulge-Removal($B, 0$)
  if *progress* > 0 **then**
    Restart loop
  end if

  progress ← Complex-Bulge-Removal(B, *Expansion*)
until *progress* $= 0$

Bulge networks will generally be a lot more complicated than just two parallel paths. We may want to revisit global approaches to simplifying them, such as the use of a spanning arborescence, rather than the local approach outlined above.

8. Working with the implicit graph representation

It is absolutely possible to work with the graph implicitly defined by the set E rather than by explicitly forming edges and vertices. Section 9 gives two other graph representations that we may use instead to improve speed, but both will require more memory.

Addressing a vertex: A vertex in E can be addressed as $v = (e, \alpha)$ where $e \in IDS$ and $\alpha = 0$ or 1. Let ID-To-Raw$(e) = e_1 \cdots e_{k+1}$. Then v is the vertex with sequence $e_1 e_2 \ldots e_k$ (when $\alpha = 0$) or $e_2 e_3 \ldots e_{k+1}$ (when $\alpha = 1$). This address is not unique.

In double-stranded mode, the dual vertex can be addressed as (Dual-ID$(e), 1 - \alpha$).

Recall that *IDS* consists of IDs of edges that are present in the edited graph, not the ones from the original graph. Do not address a vertex by (e, α) where $e \in IDS_0$ but $e \notin IDS$.

Degrees: For $v = (e, 0)$, the out-degree of v is computed by changing the last nucleotide to each of the other three possibilities and searching the list of edges in E to see if they exist. The in-degree is computed by shifting in each of the four nucleotides (one at a time) and checking to see if the resulting string exists as an edge in E_0. For $v = (e, 1)$, the computation is similar.

Paths of the branching graph of G: When $v = (e, 0)$ is a branching vertex with Out-Degree$(v) > 1$, or v is a source, we may find the path emanating from v with initial edge $e_1 = e$ as follows:

Algorithm 20 Form-Path(e)

```
P ← e
repeat
  f ← Extend-Path(last edge of P)
  if f ≠ FAIL then
    P ← P, e
  end if
until f = FAIL
```

Algorithm 21 Extend-Path(e)

```
w ← (e, 1)
if In-Degree(w) = 1 and Out-Degree(w) = 1 then
  return the unique edge extending w by one nucleotide out
else
  return FAIL
end if
```

9. Alternative graph representations

There are two proposals for making graph traversal more efficient: Forming the k-mer representation of the de Bruijn graph, using one byte for each k-mer to represent all its in/out edges (since there are at most 4 edges in, and at most 4 edges out); and forming the branching graph, in which the vertices are just the ones with in-degree or out-degree not equal to 1, and the paths represent sequences of lengths $\geq k+1$.

9.1. **Explicit k-mer graph representation.** Build a table of all canonical k-mers in *CleanReads*. For each k-mer x, store a one-byte payload:

- One bit each: are $x+\mathtt{A}$, $x+\mathtt{C}$, $x+\mathtt{G}$, $x+\mathtt{T}$ in E_0?
- One bit each: are $\mathtt{A}+x$, $\mathtt{C}+x$, $\mathtt{G}+x$, $\mathtt{T}+x$ in E_0?
- For Colorspace, use $\mathtt{0},\mathtt{1},\mathtt{2},\mathtt{3}$ instead of $\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}$.
- For other alphabets, we would not be able to do this in one byte.

For traversing the de Bruijn graph one edge at a time, it is indeed faster to use this representation than just using the table of $(k+1)$-mers, but it takes about twice as much memory. However, the condensed graph representation is even faster to traverse and takes even less memory.

9.2. **Condensed graph representation.** Earlier versions of EULER use a condensed graph representation, where each chain of 1-in/1-out vertices is replaced by a single edge representing more than $k+1$ nucleotides. Initially this is the branching graph of G_0, based on the sources, sinks, and branching vertices of G_0. However, as graph editing operations are done, the resulting graph may develop its own 1-in/1-out vertices, unless we choose to continually condense its edges.

Vertices: Form a graph G_1 whose vertices V_1 are the vertices of G_0 where

$$(\text{In-Degree}(v), \text{Out-Degree}(v)) \neq (1,1) \ .$$

This includes branching vertices, sources, and sinks.

Isolated cycles: If there are components of G_0 that are isolated directed cycles (all vertices in them have In-Degree(v) = Out-Degree$(v)=1$), then we arbitrarily choose one vertex in the component (and its dual vertex) to include in our condensed graph. Or, we can treat these components specially (delete them if they are below a certain length, or output them intact otherwise). Short isolated cycles may arise from bogus reads consisting of a tandem repeat of low complexity that is not in other reads. Longer ones should be rare, but in principle, assembling a circular genome with no repeats from perfect reads would result in a circular graph. Any repeats or imperfect reads could break the circle.

Edges: For each vertex $v \in V_1$, follow each outgoing path from it in G_0 through the 1-in/1-out chain of vertices, until reaching another vertex $w \in V_1$.

This path becomes an edge in G_1.

If the path consists of r edges in G_0, then it represents $k+r$ nucleotides.

Since edges in G_1 have variable lengths, we need to store a length in nucleotides for each edge.

Graph editing operations: We can either adapt all graph editing operations to simultaneously update G_1 along with *subs*. Or we can redesign all the algorithms to replace substitutions of $(k+1)$-mer edges in G_0, by substitutions of edges (of arbitrary lengths in nucleotides) in G_1. The latter is

probably better but may require more details to fully describe. I will probably write up pseudocode for the latter rather than the former. The latter may also be more compatible with the string graph produced by the Durbin and Simpson FM-Index paper.

Note that as we edit the graph, vertices of G_1 may become 1-in, 1-out, and we will have to account for that.

9.3. **FM-Index New section.** The FM-Index paper by Durbin and Simpson may provide another method of constructing the condensed graph, instead of explicitly constructing all k-mers or $(k+1)$-mers first. Hamid will explore this. Some questions are:

- Can the FM-Index construction be used to create the same condensed graph as we describe here (and if it's not the same, what are the differences)?
- Memory and speed of Simpson and Durbin's FM-Index implementation for the initial graph construction, compared with the corresponding phases only of EULER-SR and Velvet on the exact same datasets.
- Is it desirable to convert from the FM-Index plus Bowtie structure to an ordinary graph (or bidirected graph) structure, with node and edge objects. Or can can we (and should we) keep it all within their FM-Index plus Bowtie structure. It really depends on whether their structure allows for traversing the graph, finding in/out degrees, and supports the graph editing operations we discussed (or if not, are there any other useful graph editing operations that become easy).
- Instead of keeping k constant for all nodes while allowing condensed edge lengths to be larger, can/should we allow variable size vertices, and does this structure support that?

9.4. **Read intervals in EULER-SR New section.** EULER 2001, 2004, and EULER-SR use read intervals.

Each edge of the condensed graph has a single sequence associated to it (based on the original edges it came from before editing), and a list of read intervals. A read interval consists of

- *readID* (including whether it is primary or dual)
- *start* and *length* of interval to take from the read.
- *offset* along the edge at which to place the interval.

Characterizing its location on the edge by a single *offset* parameter is problematic.

When EULER-SR pastes edge e_1 onto e_2 together (using edge-and-vertex pasting on the condensed graph), it re-assigns all read intervals from e_1 to e_2, keeping the same offsets. But e_1 and e_2 may have different lengths. If e_1 is longer than e_2, some intervals may run over the end of e_2. And if we had pasted their duals together, the result would be inconsistent when e_1 and e_2 have different lengths, because the offsets from the vertex on the other end would place it in a different location..

Next, after it pastes edges together, a vertex v might become 1-in/1-out, say $u \xrightarrow{e_2} v \xrightarrow{e_3} w$.

In this case it condenses the edge into $u \xrightarrow{e_4} w$. The read intervals from e_2 are transferred intact. The read intervals from e_3 are transferred but with their *offset* fields increased by the length of e_2.

We need to handle this differently.

At minimum, we should give a range of potential starting positions. The following method will accurately give the range of offsets even with different length edges, and the computations in

primary and dual directions turn out to be compatible. When e_1 of length ℓ nucleotides is pasted to e_2 of length m nucleotides, positions correspond as follows:

- if $\ell = m$: every position x along e_1 corresponds to the same position x along e_2.
- If $\ell > m$: position x on e_1 corresponds to some position in the range $[x - (\ell - m), x]$ on e_2.
- If $\ell < m$: position x on e_1 corresponds to some position in the range $[x, x + (m - \ell)]$ on e_2.

The effect of this on read interval offsets is: Initially, a read interval maps to an edge with an offset $[x, x]$. Then when pasting e_1 of length ℓ to e_2 of length m, each read interval in e_1 with offset $[x, y]$ is transferred to e_2 as:

- If $\ell = m$: $[x, y]$
- If $\ell > m$: $[x - (\ell - m), y]$
- If $\ell < m$: $[x, y + (m - \ell)]$

Note that the offset is used for multiple purposes, including finding distances between paired reads in the graph, and generating SAM files. It could be used for a consensus stage (currently not directly implemented in EULER-SR, but SAMtools can use the information to give its own consensus).

9.5. **Path mapping instead of read intervals** New section**.** Consider a bulge comprised of two paths in the condensed graph:

$$\begin{aligned} P_1 &: V = T_0 \to T_1 \to \ldots \to T_r = W \\ P_2 &: W = U_0 \to U_1 \to \ldots \to U_s = W \end{aligned} \tag{3}$$

where r, s count edges in the condensed graph.

As in Section 7.2, the portion of path P_1 that we may consider pasting to a portion of P_2 is found as follows: let

$$\begin{aligned} &i \text{ be maximal in } 0 \le i < r \text{ with Out-Degree}(T_i) \ge 2 \\ &j \text{ be minimal in } 0 < j \le s \text{ with In-Degree}(T_j) \ge 2. \end{aligned} \tag{4}$$

If $i \ge j$, we cannot safely paste part of P_1 to part of P_2.

If $i < j$, we may paste subpath $Q_1 = (T_i, T_{i+1}, \ldots, T_j)$ from P_1 to P_2 using a version of edge-only pasting adapted to condensed graphs. When we do this, we need to take into account that the edges in P_1 and in P_2 may not start or end at the same positions (in nucleotides). In Paste-Subpath (Algorithm 16 line 10), interpolation is used to give an approximate location; that's probably sufficient for the full graph, but in the condensed graph, we should not paste whole edges. Perhaps the edges need to be broken up.

If we decide to allow complex pastings of this sort, we need to develop a good data structure for remapping paths to paths, and for improving on the single *offset* parameter:

- As previously described, replace *offset* by a range to allow for different lengths in source and destination, and so that positions in primary and dual directions are consistent.

  However, the previous description was for pasting one edge to one edge in the condensed graph. For pasting paths of condensed edges to paths of condensed edges, it will be more involved.

- We have to develop a way to represent chains of path remapping operations. So if path Q_1 is remapped to path R_1 (which may start in the middle of a condensed edge, continue through whole condensed edges, and end in the middle of another condensed edge), and then R_1 is remapped to S_1, etc., we have to account for that.

  The *subs* map on the full graph achieves this by operating at an atomic level. But we are discussing working with condensed edges or paths formed by condensed edges,

- When we get to the read mapping stage: Each read is mapped to a path in the original condensed graph, and parts of it then go through remappings as just sketched out, resulting in a new sequence of edges (not necessarily a contiguous path); if it's not a contiguous path, it will have to be repaired.

10. Read mapping

Graph simplification is done in a way that preserves the ability to map reads and mate pair paths to the simplified graph. However, bulges with parallel paths of different lengths, and complex bulges where we only paste part of one path to the other, may introduce some noise.

During graph simplification, the reads are stored on disk, not in memory.

After graph simplification, we process the reads by rereading them from *CleanReads* and mapping them to paths in the graph.

To map a single read to an edge-sequence in the graph:

Algorithm 22 Map-Read(r,G)

inputs
  r is a string with symbols $r_1, r_2, \ldots, r_L$
$P = \emptyset$ ▷ Edge sequence
for $i = 1$ **to** $L - k$ **do**
  $e \leftarrow$ Remap(Raw-To-ID($r_i r_{i+1} \ldots r_{i+k}$))
  if $e \neq$ DEL **and** $e \neq$ DEL' **then**
    $P \leftarrow P, e$
  end if
end for

Note that the edge-sequence P may not be a path due to mis-sized path pastings and partial path pastings:

- It may have repeated edges (due to pasting a larger path to a smaller one). This is fine; we still have sufficient information to locate the read.
- There may be gaps between edges (due to pasting a shorter path to a longer one). Again this is fine; we still have sufficient information to locate the read.
- There may be consecutive edges in the sequence that are not connected by a short directed path in G. This arises from partial path pastings. We will have to come up with a good way to fix the mappings that arise in this manner. There should be a subsequence of edges in P that can be expanded to a unique path.

New paragraph If we work in the condensed graph, the idea is similar but we will get a sequence of edges on the condensed graph. The intermediate edges will (probably) be whole. The first and last edges may be partial. We need to specify locations along these partial edges, similar to the read interval *offset* parameter or improvements on it as proposed in the previous section.

This can be generalized to strobe reads.

Let r_1, r_2 be paired reads, where we have adjusted the orientations so that they are both in the same direction with r_2 coming after r_1 at an approximately known distance.

Map r_1 to edge-sequence P_1 and r_2 to edge-sequence P_2 as described in the previous section.

If we develop a good heuristic for repairing paths damages by partial path pastings, then we will just have to look for paths in G from P_1 to P_2 of the appropriate length:

- If there are no such paths, we ignore this mate pair. Since graph simplification was designed to avoid losing paths, this usually arises from chimeric mate pairs that did not have a correct path in the original graph.

- If there is a unique such path, we declare the sequence along it to be a *mate-read*. However, it may be better to work in the alphabet of condensed edges rather than with the literal sequence.

- If there are multiple such paths, we may have to put this pair aside for now, or we may be able to extend one or both reads to the longest unique extension possible.

We should be able to develop a good way to repair paths broken by partial pastings. But if we do not develop a good heuristic, another approach to mapping read pairs is based on trying to find some pair of a $(k+1)$-mer in P_1 and a $(k+1)$-mer in P_2 that have the correct approximate distance in the graph, and use those to find a single path between r_1 and r_2. We may split up P_1 into groups of $(k+1)$-mer edges on the same branching edge of G, and likewise split up P_2. It is sufficient to search for paths of the correct length and orientation between one $(k+1)$-mer of P_1 on each condensed edge of G, and one $(k+1)$-mer of P_2 on each condensed edge of G.

A variation that Pavel prefers is to discard reads and only consider k-mers. Since this outline is geared towards $(k+1)$-mers, we shall use $(k+1)$-mers instead to illustrate this. The proposal would pair up each $(k+1)$-mer in r_1 with one in r_2. If the reads have length L, this generates $L-k$ pairs of $(k+1)$-mers. Thus the number of paths to consider grows by a factor of $L-k$. Additionally, any deviation from the expected distance will be magnified by having $L-k$ pairs exhibit this deviation.

New paragraph An additional complication we may need to deal with in mapping reads and read pairs: If the initial or final condensed edge that a read maps to has only a small number of nucleotides of the read, it is possible that it is erroneous and should be truncated off. Erosion would have caught longer read tips, but a short erroneous tip (certainly 1 nucleotide, and possibly 1 to 4, say; subject to revision) may survive because, by accident, it happens to be in other reads. The reason this is important is for read pair mapping: for pair r_1, r_2, if the end of r_1 goes down the wrong branch and it survives because it was too short to recognize as wrong, that could be a problem. This suggests that we might prioritize paths from paired reads where the last condensed edge to which r_1 maps has at least, say, 5 nucleotides of r_1 (number subject to revision) and the first condensed edge to which r_2 maps has at least 5 nucleotides of r_2.

12. Portability issues New section

12.1. **Operating system.** We are only aiming at Linux and Unix variants, including Linux, Solaris, and Mac OS X. Not Windows.

12.2. **Compiler issues.**

- We should make sure the code can be compiled both in 32-bit mode `g++ -m32 ...` and in 64-bit mode `g++ -m64 ...`. There are multiple models for how compilers handle each word size (e.g., LP64, ILP64, LLP64), and soon enough, 128 bits may succeed 64 bits. If we have code that makes too many assumptions about data type sizes, we'll have a difficult time porting it. If we write it with data size neutrality in mind from the beginning, that won't be a problem.

- `g++` has a number of options to issue warnings for non-portable constructions. We should routinely enable all such warnings (not just the default ones) and fix them correctly.

- Test on different Unix/Linux platforms. Currently EULER-SR can compile on Linux, Solaris, and Mac OS X, each in both 32 and 64 bit modes.

12.3. **Integer types.** Porting EULER-SR to 64-bits was an educational experience. Originally it used `int`, `char`, and unsigned versions of those, for most everything.

In the new assembler, appropriate integer types should be chosen for each object, and be defined in a header file with `typedef`s or classes. Then the main code should use the `typedef`s, not the explicit integer type. So even if read IDs are encoded with 64-bits, do not use `uint64_t` throughout the code; only use it in a header file. For example:

```
typedef uint64_t edgeid_t;    // Edge IDs
typedef uint64_t readid_t;    // Read IDs
typedef ssize_t  seqoffset_t; // Offset into a sequence
typedef ssize_t  vertcount_t; // For counting vertices
```

There are several categories of integer types:

- **Original C types**: `char`, `unsigned char`, `short`, `int`, `long`, and extensions such as `long long`, etc.: Characters might be ok, but avoid the rest.

- **Scalable types added on to C/C++ through libraries:** `size_t`, `ssize_t`, `off_t`, etc.

  These types are good to use, particularly when you'll be using C/C++ library calls that actually use these types rather than `int`s. Variables that give the sizes of C++ containers or indexes into them should be `size_t` or `ssize_t` (unless we know there is a smaller specific bound that will always be appropriate for all compilers and all sequencing platforms, in which case we can use a fixed precision type). Offsets into files should be `off_t`. Etc.

  In addition to being compatible with library calls, these also have an advantage over fixed precision types in that they automatically scale when recompiled on a different platform. So EULER-SR 2.0 can be compiled in 32-bit mode or 64-bit mode, and these types automatically adjust.

- **Fixed precision types, some added on through libraries:** `bool`, `int64_t`, `uint64_t`, bit fields, etc.

  These may be desirable to use as well. **However,** confine the actual use of them to header files with `typedef`s or members of class definitions. The main code should only use the `typedef` or class name.

EULER-SR 2.0 only uses types like `uint64_t` for purposes such as representing k-mers with the 2-bit per nucleotide packing. It uses `size_t` or `ssize_t` for container sizes and indexes into containers.

- **Pointers:** Of course they are necessary. However, as we have described in Section 2.2, for certain objects such as reads, vertices, and edges, there are advantages to using numerical codings other than memory pointers.

12.4. **Floating types.** Use `double` instead of `float`.

12.5. **Binary output file format compatibility.** For intermediate output files only intended to be reread by later assembler stages, rather than to be used by other applications external applications, we do not have to worry about binary format compatibility. It's fine to dump structures out as-is, with whatever type sizes, endianness, and alignment issues are particular to machine it's running on.

For files intended to be used with external software, we do have to worry about binary format compatibility. This includes 454's SFF format (but Mark has a converter to FASTA format already, and other sequencing platforms fortunately produce ASCII files); and BAM files for SAMtools. Fortunately, SAMtools can also take SAM format files (which are in ASCII) as input and convert them to BAM files.

12.6. **Parallelization.**