**Supplemental Assembly Methods**

**Preprocessing of data**. Raw bases and quality scores were generated using the Illumina pipeline, including the EAMSS filtering algorithm that assigns the quality score 2 to bases in the tail of a read that appear to be ‘untrusted’. If a data set has been generated without the EAMSS filter, we recommend that it be applied to the data before running DISCOVAR. We also recommend that quality scores not be recalibrated: DISCOVAR is designed to work with the native Illumina quality score distribution.

Reads were aligned to the hg19 human reference sequence using bwa (Li and Durbin 2009), as part of the Picard pipeline http://picard.sourceforge.net.

The DISCOVAR algorithm has three major stages:

* In the first stage, reads are error corrected, and as part of this process, pairs are closed by filling in intermediate bases. In some cases, multiple closures are allowed. The output of the first stage consists of these pair closures.
* The pair closures are now formally merged into a graph, using a minimum overlap K that depends on the distribution of pair closure sizes.
* In the second stage, a series of operations are carried out to simplify and improve the assembly graph.
* In the third stage, which applies only if a reference sequence is supplied, differences with the reference are deduced from the graph.

**Error correction**

**Precorrection 1**. This is exactly as described in the ALLPATHS-LG paper supplemental, as the module PreCorrect. For convenience we provide the description here. The module examines all 25-mers in all reads. It performs a large, external sort in such a way as to bring together for consideration all the 25-mers that agree about the initial and final 12-mers, ignoring the central base. For the each pile of 25-mers that agree about the flanking bases, and potentially disagree only about that central base, it examines the quality scores associated with the base call of the central base. Piles of fewer than 6 are ignored. For larger piles, the quality scores are summed separately for each of the four potential calls of the central base. We declare a winner among the four calls as the call having the greatest sum of quality scores, but only if the sum is 60 or greater. If we have a winner, then we can also check for losers. A loser call has no more than one call of quality 20 or more, and its sum of quality scores must be less than 1/4 that of the winner. Reads containing loser-calls are candidates for having those calls corrected to the winning call. One further round of sorting and screening applies the correction to a read only if it is well isolated from other proposed corrections: no changes are made to either call when two proposed corrections are within 12 bases of each other (i.e., when the flanking bases that built a pile are themselves suspect). Whenever a correction is adopted, the associated quality score for that call is set to 0. [See paths/FindErrorsCore.cc.][Turning off this step is clearly deleterious, but should explore further.]

**Pathing algorithm**. [For subsequent reference, need to describe Ted’s LongReadsToPaths algorithm. This is probably in our google doc.]

**Pair filling 1**. [Note random use of synonymous terms ‘close’ and ‘fill’; need to fix.] In this step, certain pairs are filled. The primary purpose of this step is to fill pairs that we can easily and unambiguously identify as having a unique closure, thereby decreasing the computational requirements of the subsequent pair filling steps. Let K = 60. Consider all the kmers in the reads, however exclude kmers that occur less than 5 times in totals in the read and their reverse complements, and for a given read, exclude all kmers that occur after such an excluded kmer. Form the unipath graph [ref] associated to these truncated reads. Suppose that both the left and right reads in a pair map completely to the same unipath. Then that unipath defines a closure for the pair, which we accept. (Determination of this uses the pathing algorithm.) [Need figure.] Closed pairs are marked as done, and are not subject to further processing *per se*, however they do participate in the subsequent correction of other pairs. For this purpose, the pair is replaced by two reads, that are obtained from the closure by taking the leftmost n1 bases (for the left read), and the reverse complement of the rightmost n2 bases (for the second read). Here n1 and n2 are the original read lengths for the pair. The quality scores are all set to 40. [See paths/long/FillPairs.cc and lines 442-457 of paths/long/LoadCorrectCore.cc.]

**Quality score lowering**. For pairs not already closed, we set each quality score to the minimum over the diameter 9 window of the quality scores centered at the given base. [Turning this off does many good things, and a few bad things. We should explore compensatory changes.]

**Precorrection 2**. This step is carried out twice, first using K = 24 and then K = 40. [Using only K = 24 appears to be deleterious, as does using only K = 40.] This step ‘corrects’ certain bases, and changes the quality scores for those bases to zero. In outline, the process works by finding the true friends of the read, which consist (in principle) of all reads that come from the exact same locus on the exact same chromosome. Then we find the consensus of these reads.

First, for each read, we find its initial friends. Such a friend read comes with a gap-free alignment to the given read. (This makes sense because the indel error rate in the data is very low.) Such an alignment is defined by an orientation and offset. Note that a read may appear multiple times as a friend, using different orientation and offset. Friend reads are defined by perfect kmer matches, however kmers occurring more than 1000 times in the reads and their reverse complements are ignored. [Presumably not quite right.] The initial friends are now formed into a stack under the given read. [Need figure.] If the stack height is ≥ 104, precorrection is not carried out. In this stack we truncate friends on the left and right so that they do not extend beyond the given read.

Next we ‘clean’ the stack. Initial friends having a Q30 difference with the founder (i.e. Q30 on both reads) are now removed from the stack. Then we remove any friends having a ‘high quality difference window’ with the founder, as follows. For each 10-base window on the founder, suppose that some friend agrees with the founder for the entirety of the window, and has quality score ≥ 10 at every base on it. Now suppose that another friend is defined on the window, has at least three differences on the window with the founder, and the sum of its quality scores at those bases is ≥ 30. Then the friend is declared to have a ‘high quality difference window’ with the founder and marked for deletion.

Finally, we form the consensus for the stack. To do this we proceed through each of its columns. For purposes of this calculation, quality scores of 1 or 2 are changed to 0.2. For each of the four calls (A,C,G,T) appearing in the column, we compute the quality score sum for that call, and also note the top quality score that appears. The call having the highest quality score sum is declared the winner. The top quality score is then deducted from the sum for each non-winner. We then test to see if the ‘victory’ should be accepted, according to the following requirements: the winner’s quality score sum must be at least 50, the winning sum must be at least 10 times the runner up sum, and the runner up sum must be at most 100. If the victory is accepted, and the winning call disagrees with the base on the given read, it is ‘corrected’ and its quality score is set to zero. [See paths/Correct1Pre.cc.]

**Pair filling 2**. This step is the same as Pair Filling 1, except that K = 80, and reads are truncated using a different method, namely each read is truncated at the first point where in consensus formation (above), the test fails. [See paths/LoadCorrectCore.cc lines 532-626.]

**Pair correction and filling**. [See CorrectPairs1.cc.] As in Precorrection 2, we form a set of initial friends for each read. For this we use K = 40. Both reads in a given pair are processed at the same time. We build stacks, this time allowing extension to the right for the left read and extension to the left for the right read. [Need figure.] If the stack height for either read is ≥ 104, precorrection is not carried out.

We next exclude low-quality pairs. First we compute the mean quality of all bases in the pair. Quality scores of 1 are lowered to 0 when we make this calculation. [This is weird but seems to help a little bit.] Then we find all the reads that appear in either the left or the right stack, and compute the mean quality of the totality of their bases, as above. If this mean exceeds the given pair mean by more than 20, pair correction and filling is not carried out.

Next we remove friends having inadequate glue to the founder. [If we use this term, need to define.] Carried out for each read in pair. For each friend, we find its maximal intervals of agreement with the founder. In such intervals, homopolymers longer than 10 are compressed to 10. To be adequately glued, after this compression, there must be an interval of agreement of length ≥ 20. [Code is FlagNoise.][Effect of turning this step off: deleterious. Note also that turning this on in the precorrection 2 step appears to make results worse.]

Next we raise quality scores within the stacks (and thus internal to pair correction and filling). For each founder and for each position on it that has quality > 0 but < 30, we consider the possibility of raising the quality to 30. To do this, we form the 11-base window centered at that position. We compute the number of friends that agree with the founder on this window and have quality ≥ 30 at the middle base. There must be at least 3 such reads to proceed. Next we look for friends that agree with the founder on the window, except not on the middle base, and have quality ≥ 30 at the middle base. If in this process we find 3 or more friends that assert some particular alternative middle base, we do nothing. Otherwise we raise the quality score at the founder position to 30. [Code is Raise1. Turning off all instances of this is deleterious.]

Now we look for motifs that might allow us to delete more friends. [Deleting this step is deleterious.] We scan 10-base nonoverlapping windows on each founder. We find all sequences that occur in the stack on this window. Suppose that the founder sequence appears at least 10 times. Consider alternative sequences that occur at least 10 times, and for which at least one friend has quality ≥ 20 at the middle base. All the friends carrying these sequences are deleted from the stack. [Code is MotifDiff.] After this we clean each stack by removing friends having a Q30 difference with the founder.

Next we compute the consensus for each stack. To do this we compute the quality score sum for each column, using the mapping 0 🡪 0.1, 1 🡪 0.2, 2 🡪 0.2. [Note slightly different than what used before. Changing this to the previous condition (in CorrectAll.cc) appears to substantially worsens results. Changing the previous condition to this one has a very slight negative effect that’s hard to assess. We should probably enlarge the Trav test and try again.] The consensus base is the base having the highest quality score sum. [Not sure what happens for ties, see ColumnConsensus1.]

Now we reverse complement the right stack and compute possible offsets for their relative position. [Code is GetOffsets1.] This involves finding possible overlaps for the two consensuses. We start by finding 8-mer matches between them. These define an initial set of offsets. For each offset, we define a score, its number of ‘bits’. This is computed by examining each subwindow of length at least 20 within the overlap, and computing the maximum number of bits for each. Subwindows contains a 40-base window having ≥ 20 mismatches are ignored. Let k be the number of mismatches in the subwindow, and n its length. Then the number of bits in the subwindow is defined to be

-(5/3)log10( BinomialSum( n, k, 3/4 ) ), where BinomialSum(n,k,p) = Σi=0k (n choose i)pi(1-p)n-i. We observe empirically that for a perfect overlap, this number is close to the length of the overlap. Offsets having < 25 bits are ignored. Offsets that imply a Q30 mismatch between the founder reads are ignored.

Next we go through a process that allows certain offsets to ‘invalidate’ certain other offsets. For this, for each offset, we first find the intervals of perfect agreement between the two consensuses. We exclude the first and last 10 bases from each such interval, and mark the two consensuses as confirmed on the remaining bases. These markings are separately tracked for each offset. Next we say that offset i invalidates offset j if offset j implies that con1[p1] ≠ con2[p2] for some p1, p2, but offset i confirms both of these positions. Finally suppose that for some i, offset i is not invalidated by any other offset, but offset i invalidates offset j. Then offset j is deleted.

Now suppose that offset j has < 40 bits, and that offset i has at least 10 more bits than it. Suppose that the ‘slope’ (bits i – bits j) / |offset i – offset j| is ≥ 2. Then we delete offset j. Note that the slope condition is designed to protect against deletion of alternative offsets that arise from tandem repeats. This completes the computation of offsets.

For each offset, we now merge the left and right stacks, creating a ‘joint’ stack. [Need figure.] In cases where a (read, orientation, offset) triple appears twice, only one is retained. [Note that this doesn’t tell us how quality score differences are resolved.] Quality scores of the two founders (now in the joint stack) are again raised, as described previously. After this we clean the joint stack by removing friends having a Q30 difference with one or both of the founders. If this removes one or both of the founders, we remove the offset from consideration.

Next, for each column in the joint stack, compute the quality score sum for each of the four possible base calls, but using only those reads whose partner is placed at least once in the stack. [Code: PairWeak1.] If the winning sum is ≥ 100, at least 10 times the runner up sum, and the runner up sum is < 100, remove any friend that disagrees with this winning call and has quality ≥ 30 there. If this removes one or both of the founders, we remove the offset from consideration. [Note that turning off this step appears to be deleterious, but not unambiguously so, and might bear further investigation.]

Now we create consensus and consensus quality scores for the joint stack. [Code: StrongConsensus1.] The consensus is computed as was the consensus for the founders. To compute the consensus quality scores, we first raise quality scores on each friend based on its agreement with the consensus, as follows. Suppose that a base on a friend and its d flanking bases on both sides agree with the consensus. Then if the quality score is greater than zero but less than 10\*log10(2d)\*0.5, we raise it to this number. [Explain the logic. Note that 0.5 is a fudge factor.] Then we compute the quality score sums for each column, using the mapping 0 🡪 0.1, 1 🡪 0.2, 2 🡪 0.2. The consensus quality score for a given column is set to Min( winning sum – runner up sum, 50 ). However we then test for inconsistency in each column, as follows. Suppose that the runner up sum is > 100 and supported by at least two Q30 bases. Then we declare the column inconsistent and set the consensus quality score to zero.

Next we ‘protect’ certain bases in the founders. We take the first 10 bases on the left. If the left founder disagrees with the consensus and has quality ≥ 20 there, we force the consensus to be that base, and also copy the quality score to the consensus. The same operation is carried out on the right. [Turning this step off has a small but slightly negative effect.]

Next if any base on a founder has quality ≥ 30 and disagrees with the consensus, we set the consensus quality to zero.

We next check for suspicious inconsistencies between the founders and the consensus. Suppose that a founder base disagrees with the consensus, but that its 5 flanking bases on both sides agree. Suppose that there are at least 3 non-founder rows in the joint stack that agree with the founder at the base and on the 5 flanking bases on both sides. Then we set the consensus quality to zero at the base. [This has a decidedly mixed result.]

We now attempt to recover conflicted columns in the joint stack, as follows. [Need to motivate this.] First let minq\_floor be 10 if there is more than one offset, else 5. Consider columns in the joint stack for which the consensus quality value is less than minq\_floor, however for which at least one founder has quality ≥ 2. If both founders have quality ≥ 2 and disagree and the difference between their quality scores is < 10, we do not consider the column. We choose the founder base having the highest quality score. [Not exactly true and not 100% clear what is done in case of a tie.] Then we delete any non-founder row that disagrees with the base and has quality ≥ 2 at the position. We again compute the consensus and consensus quality scores for the joint stack, as described above, and protect certain bases in the founder, as above.

Now we decide if the closure is accepted. There are two criteria. First the minimum consensus quality must be ≥ 10 (or in the case of a single offset, ≥ 5). Second we impose the following ‘minimum glue’ requirement. Using only intervals of perfect overlap of length ≥ 40 between the joint consensus and some stack row, it must be possible to walk from left to right along the consensus, using the intervals, and requiring that consecutive intervals overlap by ≥ 30 (or 20, for a single offset).

Now we examine all the closures (at most one per offset). In the special case where the closures are the same, except for differing homopolymer lengths at a single locus, they are all retained. [Here there is an inconsistency in the code. We report both corrected[id1] and corrected[id1p], unlike in the main case. Fixing this inconsistency is definitely deleterious.] Otherwise, we let L denote the maximal leftmost segment where all consensuses agree, and R be the maximal rightmost segment. If L = R we report L alone. Otherwise we report both L and R. [Note that always allowing two closures to be passed without changes does not appear to be a good thing.]

We next attempt to identify pairs that were not closed, but which appear to bridge gaps. To do this, first we find all 40-mers x that appear at least 5 times in the initially corrected reads and their reverse complements. If such an x lies on a closure c, so that x’s beginning lies at least 200 bases from the right end of c, we mark x as *right-extended*. An x which is not right-extended is said to *fail*. An unclosed read pair is now declared *special* if one of its reads contains a failing kmer or contains the reverse complement y of a failing kmer x, where the right end of y lies at least 200 bases from the beginning of the read. The entire pair correction and filling step is now repeated for these special read pairs, however with more lenient parameters, so as to increase the likelihood of closing the pair:

* In the exclusion of low-quality pairs, the mean is allowed to exceed the given pair mean by up to 25.
* When computing consensus, we allow quality scores of zero to be raised.
* Minimum glue is reduced to 15.
* The minq\_floor parameter is taken to be 0 if there is only one offset. [This doesn’t seem more lenient.]

This completes the error correction and pair closure part of the assembly process.

**Graph formation**. The corrected pairs are now formed into an initial assembly graph. This graph is a directed graph, whose edges are DNA sequences, and such that abutting edges overlap by K-1, where K is a constant (to be defined). We set K to be 0.18 times the median pair closure length [Using 0.15 or .20 is worse.], rounded to a fixed set of values ..., 60, 72, 80, 84, 88, 100, ... . For the data type described in this work, the typical value is around 80. Formation of the initial assembly graph uses the pathing algorithm. The initial assembly graph is exactly the unipath graph as described in (Butler et al. 2008).

Each pair closure c and its reverse complement are mapped to the graph and recorded as sequences of edges within it, which we refer to as *closure paths*. We also refer to the *median closure path length*, measured in kmers. Below, if we refer simply to *paths*, we are referring to arbitrary paths within the assembly graph.

The assembly graph data structure records counts for each observed edge sequence e1,…,en, denoted |e1,…,en|. Similarly we would write *e.g.* |e| or |ef| for the inferred sum associated with one or two edges.

The data structure also includes an ‘involution’ *i* that maps each edge to its reverse complement, and having the property that i2 is the identity map. These data structures are maintained as the assembly graph is modified. During this process, the involution *i* is allowed to deviate in two ways from the original definition:

* while *i* always maps an edge e to *a* reverse complement of e, there may be more than one such reverse complement, and thus i must pick between them;
* in cases where a component c of the graph is disjoint from its reverse complement, we allow the reverse complement to be deleted, and in such cases *i* is undefined on c.

**Graph simplification and improvement**. The assembly graph is now operated on iteratively, using several different steps that are described below. We note that in some cases the operations as described are asymmetrical, and in such cases the same operation is carried out ‘in reverse’.

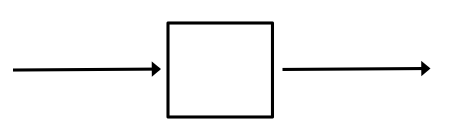
*Reverse complement removal*. When both a graph component and its reverse complement are present and distinct, one is removed.

*Hanging end removal*. First, for each vertex v, form the set of paths that start at v but contain no edge twice. This set is computed iteratively, and if at a given stage we obtain > 100 paths, the process is terminated and the computation deemed incomplete. Let D[v] be the maximum length L in kmers of all these paths. Now consider a vertex v having multiple edges ei: v 🡪 wi emanating from it. Let d[i] = L[ei] + D[wi] and order the edges so that d[0] is greatest. If for some i, d[i] ≤ 1000, d[0] ≥ 10⋅d[i], and the computation for D[wi] was complete, delete edge i. Also remove terminal loops of length ≤ 50 kmers.

*Remove small components*. Acyclic components having ≤ 1500 – K + 1 kmers are deleted, as are cyclic components having ≤ 50 kmers.

*Delete low coverage edges*. For an edge e, suppose that |e| ≤ 2, and that there is another edge f, either starting at the same vertex as e, or ending at the same vertex as e, and that |f| ≥ 5⋅|e|. Then we delete edge e. [Turning this step off is deleterious.]

*Assembly unwinding*. [Code = UnwindAssembly.] This is a method for simplifying the assembly graph. It looks for loci of the form



where the box represents ‘something between’ two edges. The method proceeds according to the following steps:

1.  We consider every assembly edge as a potential left bounding edge in the above diagram.  We then walk right from the edge (as described below), eventually defining the right bounding edge.

2.  Starting from this edge, we iteratively keep walking right, allowing for branching, until we have 10 paths or no more are possible.  We terminate a given path as soon as it contains 20 edges.  This prevents infinite looping around a circular genome.  At each iteration, we extend a path having the least number of edges in it.

3.  As we walk right, certain paths are excluded. In particular, we reject a path if it contains a subpath a,b1,...,bn,c, not seen in the closure paths at least 2.001 times, and such that the total kmer count in the bi is ≤ the median closure path length, provided that for some c’, we see a,b1,...,bn,c’ at least 5 times more often.  This criterion allows certain cycles in the assembly graph to be ‘linearized’.  The use of the median closure path length circumvents a complex statistical problem relating to how often we would expect to see a particular sequence. It is likely that more repeats can be resolved by pushing this limit.

4.  Next we look for a common truncation of the paths on the right so that they all terminate in a common edge (the right edge in the above diagram).  Having done this, we then define the ‘box’ in the middle of the above diagram.  We require that the paths traverse essentially all of its edges: if there is an untraversed edge having weight ≥ 5, we reject the box, truncate all the paths on the right, and try again.

5.  If we find a common right truncation that is acceptable, the resulting data define a recommendation, namely that the entire diagram be replaced by the set of truncated paths.  These paths still have to be squeezed back into a graph (discussed below).

6.  We prune the recommendations to eliminate conflicts.

7.  Finally, for each surviving recommendation, we squeeze its paths into a graph. Then the graph is inserted back into the main assembly graph, replacing the diagram.

*Pull apart simple branches*. Consider a diagram in which there is an edge r, with exactly two edges x1, x2 abutting r on the left, and exactly two edges y1, y2 abutting r on the right.  Let wij = |xiryj|. Suppose either that:

* w11 ≥ 2, w22 ≥ 2 and w12 = w21 = 0, or
* w11 ≥ 5, w22 ≥ 5, w12 + w21 ≤ 1 and kmer\_count(r) ≤ median closure path length, or
* w11 ≥ 10, w22 ≥ 10, w12 + w21 ≤ 2 and kmer\_count(r) ≤ median closure path length.

Then replace all five edges by two new edges z1 = x1ry1, z2 = x2ry2. [Code is PullApart. Need figure. Turning this off is deleterious.]

*Pull apart complex branches*. [Code is PullApart2. Turning it off or replacing it by PullApart is deleterious.] We look for a complex with two edges e1, e2 entering and two edges f1, f2 exiting. Let wij be the number of closure paths that contain ei…fj, i.e. ei followed eventually by fj. Then test the same criteria for the wij as in the pull apart of simple branches. If successful, we replace the entire complex of edges by all subpaths of closure paths of the form either e1…f1 or e2…f2.

*Bubble popping*. Look for edges e: v → w whose multiplicity is ≤ 2.  Then find all alternative paths from v to w whose length in kmers is within 10 of len(e), and whose multiplicity is > 2.  Suppose that there is a unique such alternative path p, that its length in kmers is within 2 of len(e), and that |p| ≥ 5⋅|e|.  Then delete edge e and replace all occurrences of it by p. [Effect of bubble popping: small and close to neutral according to Trav tests, but it might be eliminating a bunch of false bubbles that do not show up in the Trav results and could impact specificity of variant calling. Would have to look closer.]

*Graph reconstruction*. [Turning this off is deleterious. Code is WordifyAlt2.] For this method, it is convenient refer to the closure paths as words on the alphabet of edge indices. The idea of graph reconstruction is to define a collection of words that are ‘trusted’, and a collection of overlaps between them that are ‘trusted’, then formally glue the words together with each other along the trusted overlaps, yielding a new assembly. In describing graph reconstruction, we describe a general framework for pulling apart a graph assembly, together with an initial heuristic model that defines ‘trusted’.

The process starts by finding all the overlaps between all the words. Here, by an overlap, we mean a proper overlap, *i.e.* extending fully to the left on at least one word and extending fully to the right on at least one word.

The main text illustrates graph reconstruction using the example of Fig. 2. In general, with the goal of eliminating some overlaps (and in the process, also eliminating some words), we first find all subwords that occur as the overlap between two words. Each of these subwords s now seeds an analytic step, as follows. We form the matrix whose rows consist of all the words containing s. We also track the weights associated with each word. This is illustrated in Fig. 2.

For the general method, we proceed in the following complicated fashion. Let left\_exts denote the lengths in kmers of subwords appearing to the left of the seed, and let right\_exts denote the lengths on the right side. Next we traverse all possible choices for an element of left\_ext and of right\_ext. (If there would be > 1000 such pairwise choices, the calculation for the seed is terminated.) We now modify the matrix, local to the calculation for these two elements.

For each row of the matrix, walk left from the seed, counting kmers. As soon as the chosen number from left\_ext of kmers is achieved, declare the row to be left\_full, and blank out all further entries to the left. Do the same thing on the right. Call a row full if it is both left\_full and right\_full. Merge rows that are now identical, and in so doing, sum their weights. We require that the highest weight row has weight ≥ 10, and that it is full. (Otherwise the computation for the given left\_ext/right\_ext elements is terminated.) This condition establishes strong coverage at the locus.

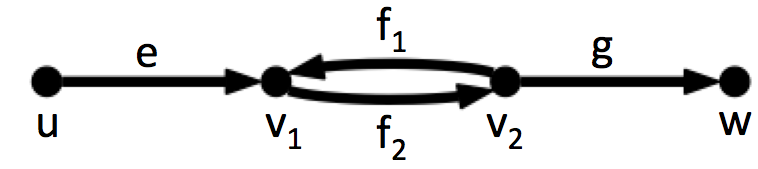
Now define potential joins. These are obtained by taking a left\_full row that stops on the right after the seed, and a right\_full row that stops on the left after the seed. Each such pair defines a join, and the unique joins (not already in the matrix) are merged in with weight zero.

Call a row a keeper if it is full and its weight is at least 2 or else it exceeds 10% of the highest weight. Call a row a follower if it is non-full and it is a subrow of a keeper.

Finally, consider non-full rows i that are not followers. Let 'total' be its weight, plus the weight of all non-full non-follower rows that are subrows of i. [Note double counting!!] Let 'control' be the maximum weight over all non-full rows whose left and right extent [define] are at least that of i. Declare i to be a loser if total ≤ 2 and 10⋅total ≤ control. From each loser row we infer that certain overlaps should be deleted, and also in some cases that certain words should be deleted. This completes the main computation of the algorithm.

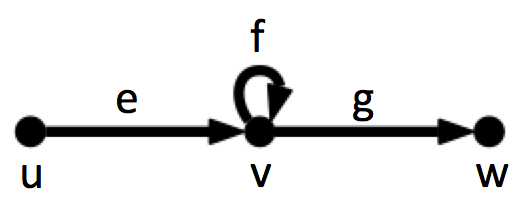
At this stage we have a collection of words and a collection of accepted overlaps between them. We first form the digraph that is the disjoint union of the words, with each word stretched out over a sequence of edges (corresponding to the letters in the word). Next from the accepted overlaps we deduce an equivalence relation on the vertices and edges of this graph. Now if the resulting quotient graph would have cases where two edges labeled by the same letter both exiting from source vertices (but not necessarily the same one), identify the two edges. Carry out the analogous operation for sink vertices. Next extend the equivalence operation by ‘zippering up’, so that the quotient graph does not have cases where identically labeled edges exit the same vertex (or enter the same vertex). Finally, form the quotient graph. This is the new graph assembly.

*Make and unroll loops*. Making loops: when we have a picture like this



replace f1 and f2 by a loop. Now unroll loops. First find loops. Loops whose length in kmers exceeds the median closed fragment length are ignored. Find the numbers c of consecutive loops (zero or more) that are observed in closure paths that include both bounding edges. Also find the numbers of consecutive loops (one or more) that occur in any closure path. If the second count set includes a larger number than the first, do nothing. Replace the loop by the linear paths defined by c.

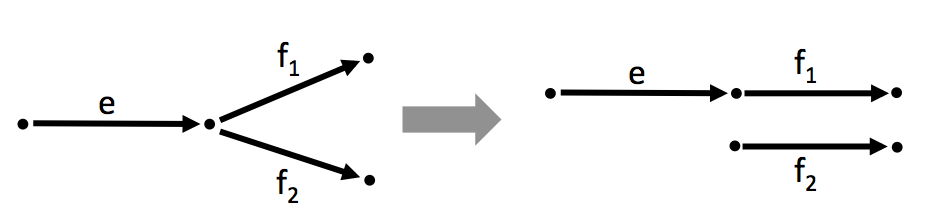
*Remove weakly supported loops*. Consider vertices v having a ‘canonical’ loop f



Suppose that |eg| > 0. If |ef| ≤ 2 and |eg| ≥ 10⋅|ef|, delete edge f. If |fg| ≤ 2 and |eg| ≥ 10⋅|fg|, delete edge f.

*Delete weakly competing edges*. [Code is KillWeakExits2.] Assign entering and exiting weights enter[e], exit[e] to edges e. Each closure path containing the edge and an edge after it contributes to the entering weight, whereas each closure path containing the edge and an edge before it contributes to the exiting weight. For each vertex v, find a neighborhood that starts from v. This neighborhood consists of all vertices within distance 4 of v, but if there are more than 20 such vertices, the computation is aborted. Consider an edge e starting from v, and edges e1 and e2, starting from vertices v1 and v2 in the neighborhood. Suppose that any path in the graph that starts from e must ultimately pass through either e1 or e2. Suppose that exit[e1] ≤ 2, enter[e] ≥ 10⋅Max( 1, exit[e1] ), and that exit[e2] ≥ Max( 1, 10⋅exit[e1] ). Then we delete edge e1.

Now consider vertices that have one edge e entering, two f1, f2 exiting. Let w[i] = |efi|, i = 1,2. Reorder if needed so that w[0] ≥ w[1]. If w[1] ≤ 2 and w[0] ≥ 10⋅Max( w[1], 2 ), separate ef1 from f2:



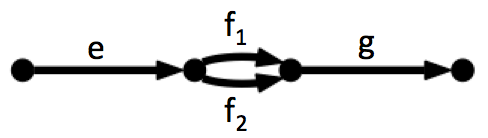
*Graph cleaning using the uncorrected reads*. [Code is DivineBubbles.] In this step we find gap-free alignments between the uncorrected reads and the assembly graph. Such an alignment goes from end to end on the read, and lands on a path of edges within the graph. It is scored by taking the sum of read quality scores > 2 at mismatches. Only alignments that begin with a 12-mer match are used. Only the alignment(s) having the best score are returned.

Define the support of an edge e to be the sum over all instances of e within an alignment as above, of 1/n, where n is the number of equal-scoring alignments of the given read.

Consider a simple branch in the graph, given by an edge e and exactly two edges f1, f2 that follow it. Suppose that support[f2] ≤ 1 and that support[f1] ≥ 5. Then we delete edge f2.

Consider each vertex v, and consider two edges e1, e2 emanating from it. We assume that neither edge starts with a homopolymer of length 10 or greater. First we find all instances of the first kmers of e1, e2 in the uncorrected reads. Then we compute the quality score sums q1, q2 associated to the last base on these first kmers. We require that q2 ≥ 100, q1 ≤ 100, and that q2 ≥ 10⋅q1. This is a prefiltering step. Next we consider not just the first kmers on e1, e2, but up to K kmers (and only as many kmers as are present in the shorter edge). On the first kmer we consider the last base; on the second kmer we consider the next to the last base, and so forth. Again we compute the quality score sums, but not counting the same position on a given read more than once. We also compute the location counts c1, c2 associated with these sums. We require that c2 ≥ 8⋅Max(1, c1), and we impose the same conditions as before on q1, q2. If all these conditions are satisfied, we delete edge e1. [This is the wingle branch approach. Turning it off is deleterious. Could wingling be done using the alignments rather than kmers?]

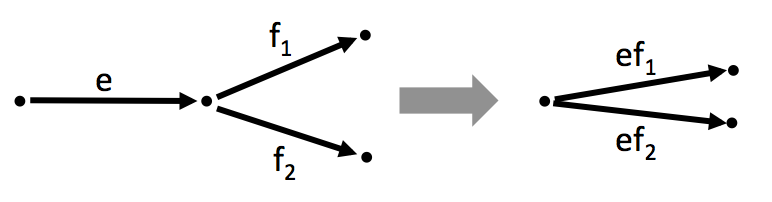
Now consider a simple bubble



We subdivide support[fi] into support in the forward and reverse orientations, defining fi and ri. [Oops, we’ve used the same notation twice, need to fix.] Let p = Min( 0.5, f1/(f1+r1) ) / 2. Let n = ⎣ f1+r1+f2+r2 ⎦. Let q = BinomialSum( n, ceil(f2), p ). If q < 10-5 and f2 + r2 < 10, delete edge f2. [Need to motivate this.][This step is observed to have no effect on the Trav tests! Perhaps should turn off.]

Now consider simple bubbles corresponding to one or more substitutions. Consider reads that place uniquely on one branch of the bubble, and score their placement on the other branch. The differences between the scores are assigned to the winning branches. These differences comprise two distributions, and we check to see if one has a 'much' larger mean than the other, and in that case delete the branch with the smaller mean. To measure the difference, we assign to each distribution the associated normal distribution that reflects our expectation of its true mean. Then we take the difference of these two normal distributions. If the mean of the difference distribution is ≥ 3 standard deviations away from zero, we deem one distribution to be much larger than the other. [Turning this step off is detrimental to rhody, and doesn’t affect our human tests.]

*Gulp edges*. Whenever we have an edge e of length ≤ 20 kmers, abutting two edges f1 and f2, replace the three edges by two edges ef1, ef2.



The opposite process is also carried out. Note that these operations do not change the semantics of the assembly graph.

*Orient assembly to reference*. If a reference is provided, reverse complement any components that appear to be in reverse orientation to the reference.