Unitigger Engineering Document

Unitigger is a component of the genome assembly pipeline developed in the Informatics Research group at Celera Genomics. Unitigger is a collection of executables and intermediate data files.

Unitigger accepts an overlap graph as input. Unitigger outputs unitigs.

1. Definition of terms.
    a. Fragments are DNA reads. Only a portion of the sequence in each read is used, a contiguous run of about 300-800 "high quality" nucleotides.
    b. Overlaps are pairs of fragments connected by one or more exact matches of sequence, within some error tolerance. Of all possible overlap types, unitigger only processes dovetail and contained overlaps.
    c. Dovetail overlaps are those that cover one fragment end on both fragments. Overlaps to just the middle of a fragment are not dovetails. Dovetail overlaps are the most useful kind of overlaps for contig building.
    d. Containment overlaps or contained fragments describe fragments that are a proper substring of some other fragment. Some contained overlaps are dovetail overlaps. As redundant information, contains are useful to consensus-building but costly to unitig-graph-building.
    e. Unitigs are unrooted layout trees of fragments. They are contigs (fragment assemblies) containing no confirmed contradictory evidence. That is, unitigs contain no fragments that also belong in other unitigs. Finished unitigs contain a list of fragments, a list of pairwise alignments between fragments, and one consensus sequence. All the fragments in a unitig must be aligned, i.e., the graph of fragments/vertices and alignments/edges is connected. All the fragments in a must unambiguously belong to this unitig and no other.
    f. Chunks are intermediate graphs used to construct unitigs. Chunks are the maximum set of fragments that form a stair-step contig – the fragments can be ordered such that each fragment extends the consensus beyond the previous. Every fragment end internal to a chunk as exactly one buddy fragment end. Formally, chunks are maximum chains with no redundant or conflicting information. In contrast, unitigs contain redundant fragments that help construct a reliable consensus.
    g. Buddy overlaps are those where both fragments are the other's best overlap.
    h. Solo fragments are those with no dovetail or containment overlaps.
    i. Hanging fragments are those with dovetail overlaps on only one side.
    j. A-Hang & B-Hang [are what????]
    k. Thru fragments are those with dovetail overlaps on both sides.
    l. Spurs are hanging fragments that cause a 1-deep branch in the chunk graph. Spurs have one dovetail overlap with some interior portion of the contig. Spurs have one 'bad' end that overlaps no other fragment. Formally, a spur is a hanging fragment whose removal would not change the thru status of any thru

fragment. See figure. Spurs are filtered from Unitigger's chunk building algorithm. For convenience, spurs are not allowed to contain other fragments. These alternate interpretations of spurs are possible.
   i. The 'bad' end of the spur is bad data and should not be included in the unitig.
   ii. The spur is the correct end of the unitig. Its 'bad' end is actually good sequence in an area of no coverage, a gap. The conflicting fragments are bad data.
   iii. The spur is the correct end of one unitig in an area of genomic polymorphism. The conflicting fragments represent another haplotype. Incomplete coverage would explain the lack of overlaps on the spur's 'bad' end.

m. Reaper is a version of FGB with special parameters for filtering inputs.
n. FGB stands for Fragment Graph Builder, a component of Unitigger.
o. CGB stands for Chunk Graph Builder, a component of Unitigger.
p. CGW stands for Chunk Graph Walker, another component of the genome assembly pipeline. CGW is also known as the Scaffolder.

2. Data flow (July 2001)
   a. A typical production run of Unitigger uses components in this order.
      i. Reaper, FGB, CGB.
      ii. Repair breakers.
      iii. Reaper, FGB, CBG again.
      iv. Bubble smoothing.
      v. Reaper, FGB, CGB again.
      vi. Consensus.
   b. Overlapper output is Unitigger input.
      i. Input file of OFG messages (overlapped fragments). By design, this file has binary and text forms, but in practice, this is a text file. [Is the FragStore used? Why/why not?]
      ii. Input file of OVL messages (overlaps). By design, this file has binary and text forms, but in practice, this is a text file. [As of May 2001, there are plans to replace this input with the OverlapStore.]
   c. Unitigger output is Scaffolder input.
      i. Unitigger output is [what?].
   d. Abstract internal data flow
      i. Unitigger takes as input a graph whose vertices represent fragments and whose edges represent overlaps.
      ii. It uses heuristics to create distinct sub-graphs each with maximal genome coverage but minimal redundant information. The sub-graphs are "chunks".
      iii. Unitigger then repopulates chunks with redundant fragments; these sub-graphs are "unitigs". Unitigger characterizes each edge in ways that are helpful to the Consensus module.
      iv. Unitigger then calls the Consensus module on each unitig.

v. Finally, Unitigger outputs a set of unitigs, each containing a consensus, a list of fragments, and list of qualified edges.

3. Functionality by module (July 2001)
    a. Reaper
        i. Filters out insignificant overlaps.
            1. This is a lossy speed optimization based on arbitrary thresholds. Empirical evidence shows that filtering introduces only 1 error in 100 million unitigs.
            2. For each fragment, it discards the least significant dovetail overlaps and the least significant containment overlaps.
            3. The cut-off criteria are defined by command-line arguments [which ones?].
            4. [Cost/benefit?]
        ii. Labels some fragments as ‘spurs’ or ‘contained’. These fragments can be safely removed from chunks even tho they get reintroduced to unitigs.
        iii. Reaper optionally writes to disk its reduced graph.
        iv. Reaper’s overall benefit was a reduction in running time from months to days.
        v. The Reaper component was added Jan 2001 by Clark Mobarry.
        vi. Reaper does not exist as a separate executable. Instead, Reaper is FGB with certain parameters. Namely, ‘**fgb –x1 –z10 –d0 –M0**’.
    b. FGB
        i. Sorts the input fragments.
            1. This is a lossless speed optimization.
            2. Ordinarily, fragments are stored in no particular order. FBG [build #1 – huh????] sorts fragments into the order they are likely to be needed by FGB, based on the overlap information. The sort transforms FGB’s random access into nearly sequential access. The co-localized data is more likely to be in a memory cache when FGB requests it.
            3. Internally, Reaper assigns each fragment a 32-bit VID, and sorts the file by VID. FGB then reassigns each fragment a new VID. Note fragments are also identified by their UID (64-bit Celera-assigned unique key) and their IID (32-bit unique key assigned by the genome assembly pipeline’s Gatekeeper, sequentially in the order received).
            4. [Cost/benefit?]
        ii. Removes redundant edges by chordal overlap removal.
            1. Reduces the edge-degree of the overlap graph.
            2. Currently uses an algorithm called triangle reduction, which was proven lossless. Future versions may use a faster heuristic called thickest edge removal. [???? Clark wrote this, but I don’t

understand: Formally uses triplet reduction, proven lossless. Currently uses path following.]

3. The resulting graph is non-chordal. This means that overlaps that could be inferred from two other overlaps were removed. (Why chord? The vertices could be arranged along a circle such that all edges connect two adjacent neighbors.)

iii. Triplet algorithm is worst case **O**($n*d^3$) and best case **O**($n*d^2$), where d = dovetail overlap degree, and n = number of fragments.

c. CGB
   i. Restores redundant edges in each chunk.
d. Consensus
   i. Builds a consensus sequence for each unitig.

4. Configuration management (July 2001)
   a. Unitigger is composed of 2 executables.
      i. FGB incorporates 2 modules.
         1. Reaper module was added Jan 2001.
         2. FGB proper.
      ii. CGB incorporates 2 modules.
         1. CBG proper.
         2. Consensus module (by Karin Remington).
      iii. Executables are generated by make in the AS directory.
   b. Unitigger source code
      i. Directory AS_CGB.
         1. Developed & maintained by Clark Mobarry.
         2. Source files are described elsewhere.
      ii. Directory AS_CNS.
         1. Developed & maintained by Karin Remington.
         2. Consensus code lives in AS_CNS.
         3. Note that Consensus code is compiled into the Scaffolder as well as Unitigger.
      iii. All source code is under CVS control.
   c. Regression test suite.
      i. [???]
   d. Documentation.
      i. There is an 8-page description of the algorithm. As of May 2001, it is somewhat out of date. The document is titled, “Celera Assembler: Prototype Chunk Graph Builder Design (version 1.0)” by Clark Mobarry.
5. Potential engineering projects
   a. The Unitigger does not exploit mate pair data. Clark feels it is an open question whether it should. Note that CGW uses mate pair data immediately downstream of Unitigger.
   b. Replace the input OFG (fragment) messages with the FragStore, developed by Saul Kravitz.

        i. [Cost/benefit?]

c. Replace the input OVL (overlay) messages with the OverlayStore, now being developed by Art Delcher.
    i. [Cost/benefit?]
d. Split Reaper and FGB into separate executables.
    i. [Cost/benefit?]
e. Rewrite Unitigger's internal store.
    i. Unitigger currently uses its own directory of data files.
        1. The Reaper, FGB, and CGB modules use this store.
        2. The existing internal store is specified in the methods.h and store.h headers.The internal store wastes disk space by duplicating information from Unitigger's inputs and outputs. [How much disk space?]
        3. The internal store was developed ad-hoc. It bears no resemblance to the FragStore or the OverlayStore.
    ii. The ideal UnitigStore would extend the OverlapStore so as not to duplicate fragment information.
    iii. [Cost/benefit?]
f. Insure portability from Alpha to Intel.
g. Simplify the build process.
h. Introduce a reporting module. Replace ad-hoc logging with a common call whose verbosity can be set at run-time. Currently, all logging is done with calls to fprintf(stderr). These calls are everywhere, even in the store.c utility.

Unitigger Source Code Files (July 2001)

AS_CGB_all.h
AS_CGB_methods.h
AS_CGB_store.c
AS_CGB_store.h

These files define the persistent data store used by all components of Unitigger. These files are common to the FGB and CGB components of Unitigger; their 'CGB' prefix is misleading.

AS_CGB_io.c
AS_CGB_io.h
AS_FGB_io.c
AS_FGB_io.h

These files implement Proto-I/O. That is, they write text or binary message files.

The code design sought to abstract the underlying storage format. There is one temporary violation of the store's design. FBG_io.c contains a memcpy that exploits the structure of the underlying file. Clark intends to repair this.

AS_CGB_repair_breakers.c
AS_CGB_breakers.c
AS_CGB_breakers.h
check_breakers.c

These are the sources for a separate executable, known as Repair Breakers.

AS_FGB_buddy.c
AS_FGB_buddy.h
AS_FGB_contained.c
AS_FGB_contained.h
AS_FGG_hanging_fragment.c
AS_FGG_hanging_fragment.h

These files are specific to the Reaper executable. The Reaper identifies 'contained' fragments and 'spur' fragments. The phrase 'hanging fragment' refers to a spur.

AS_CGB_fga.c
AS_CGB_fga.h
AS_CGB_fgb.c
AS_CGB_fgb.h

These are part of the FGB executable. Their prefixes are misleading.

AS_CGB_blizzard.c
AS_CGB_blizzard.h

Deprecated. The blizzard algorithm was used as preparation for the human genome publication. It can be invoked by –Z on the command-line.

AS_CGB_fom2uom.c

Deprecated. The main() for a separate executable that produced special output for CGW that CGW no longer needs.

make_OFG_from_FragStore.c

A separate executable. A data transform utility.

Algorithm (July 2001, two-pass version)

```
unitigger () {
        fgb_cgb()
        repair_breakers()
        fgb_cgb()
        bubble_smoothing()
        fgb_cgb()
        consensus()
}
repair_breakers() {
}
bubble_smoothing() {
}
fgb_cgb() {
        reaper_pass1()
        if (experimental)
                reaper_pass2()
        else
                fgb()
        cbg()
}
reaper_pass1() {
        // Step 1.
        // Scan the input: overlaps from overlap store.
        // Build a graph in memory where
        // nodes are fragment ends, edges are directed overlaps.
        // Note graph has 2 edges for every overlap.
        // In-memory graph is a reduced representation
        // so as to fit in memory for FGB.

        // Keep X thickest dovetail edges per fragment end.
        // Note thickest overlap == smallest A-hang.
        // X is parameter (command line –xX), usually set to 1.

        //  Keep Z snuggest containments per fragment end.
        // Note snuggest containment == smallest A-hang.
        // Z is parameter (command line –zZ), usually set to 10.

        foreach (fragment end in graph)
                // Thickest edges
                sort (outgoing dovetail edges by A-hang)
                for first X (edges with smallest A-hang)
                        add edge to graph
                // snuggest containments
                sort (from-contained overlaps) [huh????]
                for first Z (edges with smallest A-hang)
                        add edge to graph

        // Step 2.
        // This is optional – not currently implemented.
        // Include all anti-parallel edges in the graph.
        foreach (edge in graph)
                load the anti-parallel edge
                add it to the graph

        // Step 3.
        // Mark frags as THRU|HANG|SOLO.
        foreach (fragment in graph)
                p = get raw dovetail degree (proximal)
                d = get raw dovetail degree (distal)
                if (p>0 and d>0)
                        label fragment as THRU
                else if (p>0 or d>0)
                        label fragment as HANG
                else
                        label fragment as SOLO

        // Step 4.
        // Mark frags as SPUR.
        // Spur fragments have one end overlapping the interior
        // of a chain, and the other end overlapping nothing.
        // Thru fragments have at both ends dovetail overlaps
        // that contribute to a chain.
        foreach (thru fragment end in graph)
                foreach (outgoing edge)
                        increment other frag's thru frag in-degree
        foreach (fragment in graph)
                if (WHAT????)
                        mark fragment SPUR

        // Step 5.
        // Mark frags as CONTAINED.
        foreach (fragment in graph)
                if (WHAT????)
                        mark fragment CONTAINED

        reap_core()
```

```
}
reap_core() {
    // “Compute thru in-degree of fragment ends”
    foreach (fragment end)
        thru in-degree = ?
    // “Gather thru in-degree of distal end of overlaps from HANG frags”
    // “Classify some HANG frags as SPUR”
    foreach (fragment)
        if (fragment is HANG)
            if (?)
                label fragment as SPUR
    // “Gather spur flags at distal from-contained edges”
    // “to turn off from-contained overlaps”
    foreach (fragment end)
        ?
    // “Classify some fragments as contained.”
    foreach (fragment)
        if ( ? )
            label fragment CONTAINED
    // “Save frags, discard edges”
    foreach (fragment)
        save to disk // don’t save edges
}
reaper_pass2 () {
    // “Find thickest dovetail edges per fragment end”
    // “modulo spur & contained flags”
    foreach (fragment end)
        if (fragment not SPUR and not CONTAINED)
            find thickest dovetail edge
            label that edge
    reap_core()
}
fgb () {
    // “Reorder graph for locality”
    // “Classify overlaps as CHORD (to be removed)”
    // “or transitively inferable”
}
```

Annotation of Reaper modules/functions (July 2001)

Biggest problem: profusion of flags arising from persistent special case approach.

<u>create_afr_to_avx()</u>

Assigns VID to each fragment. VID is a sequential number {0..count}, as ordered by get_iid_fragment() => VAgetaccess(). [What is the use of renumbering in arbitrary order?] Returns array of IID to VID.

Array is malloc'd to size max IID. [This seems excessive for small genomes – on pyro this function consumes 66% of time.] Array is used by many functions. Array is free'd by main().

Several asserts. No dependencies. Some questionable casts of IDs to int, especially inside #ifdef _OPENMP. [How much gain in parallelizing this step?]

<u>add_overlap_to_graph()</u>

Given an overlap, adds zero or two edges to graph. Performs tests: overlap must not be dovetail to spur or dovetail to contained.

Makes many asserts. Keeps counters. Handles reflection of containment edges slightly differently ("granger edges"). Receives too many parameters. Some parameters (the rules) are not used. Asserts tons of stuff and printfs lots of diagnostics. Core code is near-identically cut&pasted 4 times.

This should be a method on the graph. Increments counters, passed from above as parameters. Statistics keeping should be a graph-object task.

<u>Insert_Aedge_into_the_edge_array_wrapper()</u>

Called by add_overlap_to_graph(). Parameters: frags, edges, 2 thresholds for end degree. The param, nedges_delta, gets passed all around, but it just counts edges added (incremented in Insert_Aedge_into_the_edge_array_simple()). The param, next_edge_obj is also passed all around, finally to GetVA_ and EnableRangeVA_IntEdge_ID(); what is it for?

Two cases are handled with near-identical cut & pasted code blocks.

Edges to high-degree fragment ends are not added!

If edge degree < threshold: 1. Add the edge. 2. Increment fragment.degree. For dovetail overlaps, this is nsuffix_dvt or nprefix_dvt, depending on asx. [Abuse of variable?] For containment overlaps, this is nsuffix_frc or nprefix_frc, depending on asx. 3, Set the frag.segend_suffix (or prefix, depending on asx) to edge ID. 4. Set next_edge_obj; why??? 5. Add to keep_a_list_of_extreme_elements().

<u>add_to_histogram()</u>

Increments counters by bucket, maintains min & max. This function accepts a histogram as parameter, and the histogram contains function pointers.

Should be method on a generic histogram utility object, possibly commercial, possibly shared with other subsystems.

Was called 388,000 times on 61,000 fragments. Why? Referenced by many files. In this run, called by 3 programs.

Contains assertions. Optional utilities should not assert.

process_one_ovl_file()

Does file I/O, then calls input_phase(), which calls input_mesgs(), which just reads messages. All data structures are passed down.

Rest is #ifdef'd out (but cut & pasted elsewhere). REAPER_MODE is always defined (by AS_CGB_all.h).

process_ovl_store()

For each OVL from stream, add overlap to graph. Calls add_Long_Olap_Data_t_to_graph(), which is a filter before add_overlap_to_graph().

Short. Takes too many parameters. Some (the rules) are passed on to lower levels, but never actually get used.

Convert_Olap_To_Aedge()

Called by process_ovl_store(). Sets edge flags based on overlap a_hang and b_hang, where improper means both are negative.

- avx = olap.a_iid (the A-frag)
- bvx = olap.b_iid (the B-frag)
- ahg = olap.a_hang (or –b_hang if improper)
- amn = ahg
- amx = ahg
- bhg = olap.b_hang (or –a_hang if improper)
- bmn = bhg
- bmx = bhg
- asx = proper
- bsx = !improper ^ !flipped   (huh?)
- quality = ExpandQuality(olap.erate) [FRAGMENT_CORRECTION compiler switch determines if original or corrected error rate is used. Boo.]
- invalid = FALSE
- reflected = FALSE
- nes = INTERCHUNK or TO_CONTAINED or FROM_CONTAINED

add_Long_Olap_Data_t_to_graph()

This is a filter before add_overlap_to_graph(). Does not add overlap if error threshold is exceeded.

This should be part of an overlap filter object.

There are too many parameters. Some (the rules) are passed on but never actually get used.

processing_phase_3()

Post-hoc analysis. Calls write_fgb_store() [checkpoint?]. Reports memory usage. Optionally writes statistical report. According to gprof, nearly all its time is spent in fragment_graph_analysis().

<u>identify_early_spur_fragments()</u>

Parameters are frags & edges (arrays). Malloc and free 2 arrays of int-counter for each frag (should frag store these counters?). Initialize to 0.

Contains unnecessary function calls inside loops (e.g. get_asx_edge()) where code blocks were cut & pasted. Contains (old?) code that stores single bits in IntFragment_ID (e.g. asx and bsx); this makes it look like every edge has 6 associated fragment IDs!!! Also gets the fragment ID (avx) from the edge, then uses it to get the (same?) fragment ID from the fragment.

1. Accumulate sums of thru fragment prefix & suffix dovetail overlap counts. “Find incoming dovetail degree from thru fragments.” For each edge, if edge.nes is INTERCHUNK or TRANSCHUNK, and if edge.avx-fragment.label is THRU, then increment counter (suffix count if edge.bsx true, else prefix count).

2. Mark spurs. “Hanging fragments will be removed unless redeemed.” For each fragment, if frag.lab is HANGING (includes HANGING, HANGING_CRAPPY, HANGING_CHUNK), then set frag.label=HANGING_CRAPPY and set frag.spur=TRUE. Why wasn’t this done earlier????

3. Unmark spurs. “For each hanging fragment, check overlapping frags. If their incoming dovetail degree from thru frags is zero, then this frag is not a spur.” For each edge, if edge.nes is INTERCHUNK or TRANSCHUNK, and if edge.avx-frag.spur is TRUE or edge.avx-frag.label is HANGING_CRAPPY, and if non-zero edg.bvx-frag prefix count (or suffix count if edge.bsx true), then set edge.avx-frag.label=HANGING and set edge.avx-frag.spur=FALSE.

<u>contained_fragment_marking_frc()</u>

Initialize all frag.contained=FALSE. For each edge, if edge.nes is FROM_CONTAINED, and if not (edge.bvx-frag.label is HANGING_CRAPPY and edge.bvx-frag is spur), and if not deleted (edge.avx-frag or edge.bvx-frag), then set edge.avx-frag.contained=TRUE and set edge.avx-frag.label=UNPLACEDCONT.

<u>separate_fragments_as_solo_hanging_thru()</u>

For every fragment, possibly set frag.label to SOLO, HANGING or THRU. Lots of printf() and assert() on running totals.

Much depends on #ifdef USE_REAPERS_DVT_DEGREE, always TRUE. What did it mean?

Data structure Aedge (AS_CGB_methods.h):

- IntFragment_ID avx,bvx;
- int16 amn,amx,bmn,bmx;
- float32 quality; “Zero is perfect overlap”
- int16 tmp16;

- int8 nes; “The edge labeling”
- bit asx,bsx,reflected,invalid,bit4,bit5,bit6,bit7;

Data structure Afragment (AS_CGB_methods.h)

- Fragment_ID uid;   // Clark, are these really in memory ?
- IntFragment_ID iid ;
- bit deleted; set_del_fragment() called when message.action=AS_DELETE
- bits contained; set_con_fragment()
- bit spur;
- int32 nsuffix_dvt, nprefix_dvt ; “Length of pref/suffix dovetail edges” actually holds dovetail overlap degree
- int32 nsuffix_frc, nprefix_frc ; “Length of pref/suffix containment edges” actually holds dovetail overlap degree
- lots more…

Reaper Rewrite: Project Goals (August 2001)

Unitigger is one stage of Celera's WGS assembler. Unitigger consists of 4 components: FGB with reaper options, FGB without reaper options, repair breakers, CGB. The Unitigger components operate in repetitious serial runs as determined by a script. The order of repetition is still experimental and subject to change. The reaper step preprocesses overlaps and heuristically identifies the most important ones. Reaper reduces the problem complexity for downstream Unitigger components, namely FGB.

This project aims to rewrite the Reaper as a stand-alone component. This project may break ground for a larger rewrite of the WGS assembler.

Goal 1. Increase maintainability.

- a. This is the main driver for this exercise.
- b. Existing reaper code is probably not maintainable by anyone except its author. The code evolved during wrenching algorithmic changes, and always under time pressure. Today's code is complex and not modular. The code handles many special cases.
- c. This is a good time to reinvent unitigger code, because a new algorithm is being deployed. Previous unitiggers have used time-costly algorithms (triangle reduction or incremental path following) to reduce the input graph complexity. The new unitigger will use an experimental linear-time algorithm (thickest edge reduction) during its Reaper pass.
- d. The new code should be modular as per object-oriented design principles. Control flow should be short (~300 lines). The code should operate generally, avoiding special cases. The new code must satisfy OOD criteria to be established by Jason Miller.
- e. The Assembly Team should review the new design and code at several milestones.

Goal 2. Keep it simple.

- a. Just filter overlaps while streaming through them one-at-a-time.
- b. Within the one-at-a-time constraint, perform all possible overlap pre-processing operations.
- c. Defer to FGB all tasks that require global overlap information, edge following, or an in-memory graph representation. Specifically, defer these tasks.
  - i. To identify spur fragments and generate statistics.
  - ii. To identify buddy overlaps and generate statistics.
  - iii. To identify non-spur contains and generate statistics.
  - iv. To identify inter-chunk overlaps and generate statistics.
  - v. To convert the implicit graph from OVL style to FGB style.

Goal 3. Validate correctness.

- a. Output statistics that will help detect problems.
- b. Output diagnostics that will help explain and fix problems.
- c. Output information useful to the new assembly comparison project.

   d. Satisfy correctness criteria to be established by Clark Mobarry.

Goal 4. Time & space.

   a. Consume under 4GB RAM.
   b. Execute quickly. Prefer sequential-access I/O over random-access.
   c. Satisfy time & space requirements to be established by Granger Sutton.

Goal 5. Rationalize storage.

   a. Today's scheme uses multiple formats. Components variously read the ovlStore format, various message formats (OFG, OVL, ADT), and the fgbStore containing the FGB snapshot format. This complexity should be reduced, if possible.
   b. The FGB snapshot format offers the highest I/O performance (bulk memory dumps), but forces binary memory compatibility between components. This tight coupling should be reduced, if possible.
   c. Today's scheme is single-threaded. All processing halts during lengthy bulk I/O operations. I/O wait time should be reduced, if possible. For instance, a threaded program could use simultaneous background database updating.
   d. It is a Unitigger requirement that every input fragment be output in exactly one unitig. Thus, Unitigger should produce a forward mapping of fragments to unitigs. Toward that goal, Reaper should produce a forward mapping of input fragments to output fragments. An error-prone implementation would rely on union operations across a multiplicity of files. A better implementation might log all activity to a fragment-history table in a database. (Note that iterative unitigging is envisioned, whereby progressively smaller lists of active fragments get processed per iteration. This will increase the danger of the multiple files approach.)

Goal 6. Maintain compatibility.

   a. Reaper must remain capable of obtaining overlaps from the ovlStore (Aug 2001 format).
   b. FGB must remain capable of reading Reaper's output. FGB is the Unitigger component down-steam of Reaper. FGB could be modified to read a new format. Currently, FGB reads either of these formats:
      i. Combination ovlStore and messages (OFG, ADT, OVL).
      ii. A Unitigger checkpoint file, containing a binary memory dump of VAs (variable length arrays).

Reaper Rewrite: Functional Specification (August 2001)

Function 1. Read inputs.

a. Fragments.
   i. On pass 1, these consist of fragment IID, fragment length.
   ii. Subsequently, these additional fields: active(y/n), spur(y/n), contained(y/n), repetitive(y/n), maps to multiple chunks(y/n), mapping to single chunk(chunkIID, 5'coord, 3'cood).
   iii. [Do we represent fragment ends separately?]

b. Overlaps.
   i. On pass 1, these consist of what the ovlStore has: implicit proximal fragment IID, distal fragment IID, ahang, bhang, bflip(y/n), quality measure.
   ii. Subsequently, these fields: proximal fragment IID & end, distal fragment IID & end, [verify this:] aflip(y/n), bflip(y/n), buddy(y/n), thickest(y/n), containment(y/n), backbone(y/n).

Function 2. Transform overlaps from ovlStore.

a. The ovlStore uses a compact notation that is inappropriate for unitigging.
b. In ovlStore, the 'A' fragment is oriented 5' to 3', and the 'B' fragment may or may not be flipped. For dovetails, positive ahang mean B's left is right of A's left, and negative ahang means B's left is left of A's left.There are 2 records per overlap (A to B and B to A).
c. For Unitigger, both 'A' and 'B' fragments are candidates to be flipped. For dovetails, ahang is always positive (if was negative, flip the A frag). There are 2 records per dovetail but only one record per containment.

Function 3. Mark and filter based on local criteria (no edge following).

a. Mark dovetail overlaps. Distinguish and mark dovetail overlaps vs. containment overlaps. The distinction is implicit in the ovlStore's hang values.
b. Validate every overlap for correctness of data format. Abort if errors exceed a threshold *e*.
c. Mark every fragment with its pre-reaper sums (see criteria for output).
d. Retain only containment-from overlaps (edge from containee to container). Ignore all containment-to overlaps.
e. Retain only active fragments. 'Active' means needing to be unitigged in this run. Commonly, active fragments will be all fragments in the frgStore that were not deleted by a Pre-Assembly AS_DELETE message. Alternatively, active fragments could be a hand-chosen set, or a set previously spit out as non-unitiggable at higher stringency.
f. Retain only overlaps between active fragments. If either fragment in an overlap is not in the active list, disregard that overlap.
g. Retain only high quality overlaps. Enforce a quality threshold *q*.

Function 4. Mark and filter based on neighborhood criteria (one-hop edge following).

a. Mark buddy overlaps.
b. Mark and remove spur fragments.

c. Mark and remove contained fragments. (Contained by non-removed, non-spur fragments).

Function 5. Apply the minimum a-hang" rules.

a. Remove thin dovetail overlaps. Discard all but the *d* minimum a-hang outgoing dovetail overlaps per fragment end. This is the "thickest edge heuristic", or the "minimum a-hang rule for dovetails".
b. Remove thin containment-from overlaps. Discard all but the *c* minimum a-hang outgoing containment-from overlaps per fragment end. This is the "snuggest container heuristic", or the "minimum a-hang rule for containment."
c. In the case of a tie, retain all the tied overlaps.
    i. Tie needs to be better defined.
    ii. Regrettably, this solution may force downstream programs to use variable length data structures.
    iii. Any solution must be deterministic. Ties should be resolved identically during different runs on the same data.

Function 6. Write outputs.

a. Fragments.
    i. Write all fragments that were read. This includes solo (zero dovetail overlap) fragments. Write one record per fragment (not one per fragment end).
    ii. Write fragment IID, prefix raw dovetail out-degree, suffix raw dovetail out-degree, raw containment-from degree, and raw containment-to degree. Raw means pre-reaper.
b. Overlaps.
    i. Write only those overlaps not removed by reaper.
    ii. Maintain CGB-style hangs.
    iii. Otherwise, write overlap records, same as input.
c. Statistics. These will help compare assemblies.
d. Diagnostics, or audit trail. These will help debug problems.

Function 7. Points of clarification.

a. Reaper retains the thickest overlaps for <u>both</u> ends of every fragment. This is true even when both ends overlap the same fragment, as in equivalence (mutual containment) overlaps.
b. Reaper reads in-edges and out-edges, but it writes only out-edges. It discards in-edges but accumulates in-edge totals per fragment end.
c. Every overlap is described by two entries (A to B, and B to A). For many overlaps, Reaper will retain only one of the 2 directed overlap records. This is because most overlap pairs are not buddies (buddies each have their thickest edge to the other).
d. Reaper retains the most potentially contradictory information. Thus, the two ends of a contained fragment should usually point to different container fragments. Specifically, when 2 or more fragments contain fragment A, the two ends of A should retain overlaps to the maximally separated container fragments.

e. Reaper retains only one from-containment overlap per fragment end. In the case of multiple containment, Reaper retains the ‘snuggest’, or most negative a-hang. An alternative strategy would retain a ‘wedding cake’ relationship among serially contained fragments; this has been considered and rejected.

Reaper Rewrite: Architectural Design (August 2001)

Design 1. Write all new source code.

  a. Language: C++.
  b. Directory: AS_RPR.

Design 2. Deploy two separate executables.

  a. Fragment Reaper. Accumulate per-frag raw degrees.
     i. Stream thru overlaps. For each fragment, accumulate various raw out-degree sums.
     ii. If both inputs (fragment, overlap) are sorted by fragment IID, then practically nothing resides in memory. Process one fragment at a time: read it, read its overlaps, write it.
     iii. If both inputs are not sorted, then load all fragments into memory. Stream through the overlaps, accumulating per-frag sums. Finally, write the frags.
     iv. The fragment input list can be partitioned so program executes as parallel subtasks. To avoid database or disk contention, each task could write a separate file.
     v. Processing can execute in parallel with Overlap Reaper.
     vi. Space requirement: **O**(degree) on sorted inputs, **O**(frags) otherwise.
     vii. Time requirement: **O**(overlaps+fragments).
  b. Overlap Reaper. Reduce the overlap count.
     i. Assume ovlStore is sorted by proximal fragment IID.
     ii. Stream thru chunks of overlaps, where each chunk is all the outgoing overlaps from one fragment. Validate the overlaps. Mark the overlaps. Write out only the reaped set of overlaps.
     iii. The overlap input list can be partitioned so Reaper executes as parallel subtasks. Each partition must occur at a fragment boundary. To avoid database or disk contention, each task could write a separate file.
     iv. Note it will independently process both directed halves of each overlap, because it will encounter each half separately.
     v. Processing can execute in parallel with Fragment Reaper.
     vi. Space requirement: **O**(degree) on sorted inputs, **O**(frags+overlaps) otherwise.
     vii. Time requirement: **O**(overlaps+fragments).

Design 3. Input data formats.

  a. Fragments.
     i. The fragment input will be in a file. Usually, some preprocess will produce the file by pulling non-deletes from the frgStore. Occasionally, users will produce the file from another source (for subsetting or iterative runs).
     ii. Defer functionality to read the frgStore directly.
  b. Overlaps.
     i. Read overlaps from the ovlStore (as it exists Aug 2001).
     ii. The ovlStore has these properties.

1. For each fragment A, the store has a contiguous exhaustive list of the overlaps with other fragments.
2. If fragment A dovetails fragment B, then A's list contains an entry from A to B. Note B's list contains an entry from B to A.
3. If fragment A is contained in fragment B, then A's list contains a "containment-from" entry from A to B. Note, B's list contains a "containment-to" entry back to A.

iii. Transform ovlStore-style hangs to CGB-style hangs. (There is an issue of whether to 'Granger' containment overlaps.)
iv. Defer functionality to read OVL message files.
v. Defer functionality to read older versions of the ovlStore.

Design 4. Output data formats.

a. Write to a database interface. Abstract the storage mechanism. This should permit experimentation with various underlying implementations (messages, VA-based stores, DBMS).
b. Plan to extend Reaper's DB interface to all Unitigger components.
c. For an underlying implementation, consider BerkeleyDB. It is a high-performance, file-based database offering shared buffers and a multi-language API.
d. For compatibility with FGB, write an FGB module to read Reapers's DB.
e. For short-term compatibility with Consensus, write a dump utility to convert Reapers's DB to message files.

`Written by Jason Miller & Clark Mobarry, May-August 2001`