# Overlap Detector Module: Specification and Design

Art. Delcher

*9 March, 2000 Version 1.10*

## 1. Overview

The Overlap Detector Module of the Assembly Subsystem has two main functions: maintain a database of fragment information, and detect which pairs of fragments contain matching data. The purpose of the fragment database is to store previously received fragments, so that their matches with newer fragments can be computed. It also stores branch-point information computed by the overlapper.

### 1.1. Basic Terminology

Two sequences, *A* and *B*, are said to *overlap* iff there is a sufficiently long subsequence of *A* that matches a subsequence of *B* to within a specified degree of similarity. This degree of similarity generally reflects the belief that both subsequences were obtained from the same position in the genome being sequenced.

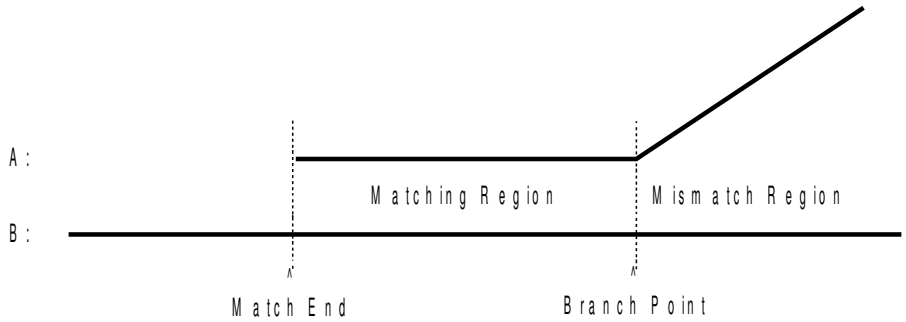
Several types of overlaps are possible between fragments *A* and *B*:

1. *Dovetail Overlap*: A complete suffix of *A* matches a complete prefix of *B*.

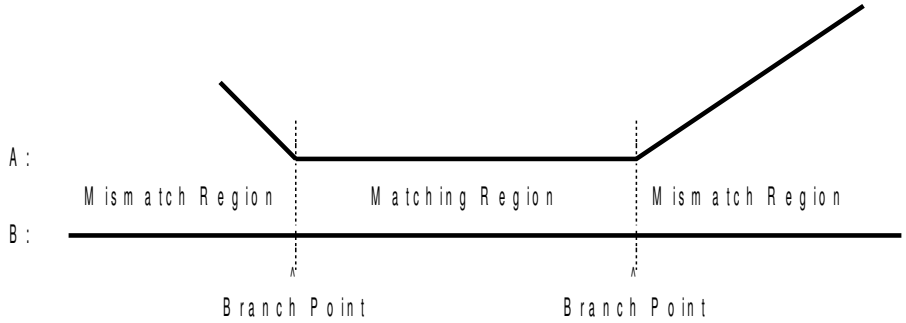

or, *vice versa*, a suffix of *B* matches a prefix of *A*.

2. *Containment Overlap*: The entirety of *B* matches a subsequence of *A*, or *vice versa*.


3. *Branch Overlap*: The matching region does not extend to the end of the fragment. Such fragments should not be combined in the final assembly. The *branch point*, *i.e.*, the boundary between the matching region and the non-matching region, indicates the boundary of a repeat region in the genome.
   a) A single branch consists of 1 matching region and 1 mismatching region.

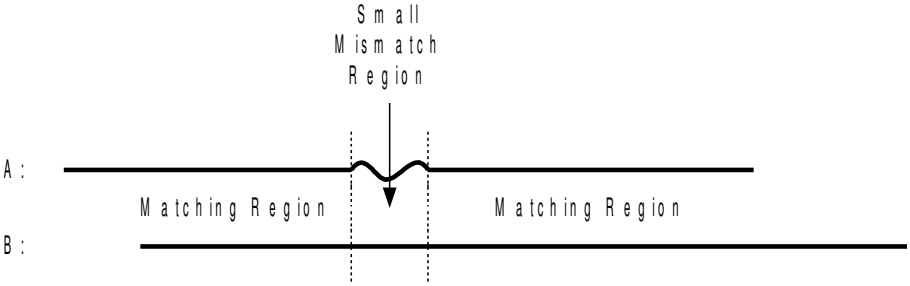



b) A multiple branch contains at least two separate matching or mismatching regions.




Branch overlaps are not included in the overlaps output by the overlap detector. The branch points from such overlaps are what is output.
Only information from simple branch overlaps is output by the overlap detector—multiple branches are ignored.

4. *Polymorphic Overlap*: An overlap that would be a dovetail or containment overlap except for a region of mismatch internal to the matching region.




The mismatch also can be an insertion or deletion. Each of the matching regions must contain a sufficiently good match to qualify as a valid overlap in its own right. More than one mismatch region is allowed.

# 2. Input/Output Specifications

## 2.1.Input

Input to the Overlap Detector Module will be a stream of messages, received from the Ubiquitous Repeat Detector Module. The messages that the Overlap Module processes are `ScreenedFragMesg`'s (`SFG`'s). The information in these messages is stored in the fragment database for later use and for checkpointing purposes. A stripped-down version without sequence or quality values is output as an `OverlapFragMesg`. Other messages are passed along unmodified to the output message stream.

The information in `DistanceMesg`'s is also stored in the fragment database before these messages are passed along.

The response to a `ScreenedFragMesg` depends on the value of the `Action` field.

```
ScreenedFragMesg:                                    {SFG
record
 action:    scalar (AS_ADD,AS_DELETE)                act:[AD]
 eaccession: Fragment_ID
 iaccession: IntFrag_ID                               acc:(%d,%d)
 variant of action:
  AS_ADD:
   record
    type:  scalar (AS_READ,AS_GUIDE)                  typ:[RG]
    source:     "description of data source"          src:↵(%[^\n]↵)*.
    entry_time: time_t                                etm:%d
    clear_rng:  SeqInterval                           clr:%d,%d
    sequence:   string(char)                          seq:↵(%[^\n]↵)*.
    quality:    string(bytes)                         qlt:↵(%[^\n]↵)*.
    screened:   sorted list of ScreenMatch            scn:↵(<SMA-record>↵)*.
   end
 end
end                                                  }

SeqInterval: record bgn, end: int end
```

When the `action` value is `AS_ADD`, all other information in the record will be stored in the Fragment Store. If the `action` value is `AS_DELETE`, the corresponding entry in the Fragment Store will be marked as deleted. The deletion entry will include a timestamp to indicate when the deletion occurred. In addition, any branch-point information that was introduced into the Fragment Store by the deleted fragment must be removed, and output messages that reflect any resulting changes will be generated.

For either `AS_ADD` or `AS_DELETE` actions, the message is emitted to the output stream as an `OverlapFragMesg`, but without sequence or quality data, to conserve space.

In addition to being added to the Fragment Store, all incoming fragments will be checked for overlaps with all other fragments in the Fragment Store. The results will be sent to the output stream as `OverlapMesg`'s and `BranchMesg`'s described below.

## 2.2.Output

In addition to updating the Fragment Store, the Overlap Module will output two types of messages describing matching regions between fragments: `OverlapMesg`'s and `BranchMesg`'s.

### 2.2.1 Simple Messages

The following is essentially a copy of the corresponding input message, with the space-consuming sequence and quality information removed.

```
OverlapFragMesg:                                    {OFG
record
 action:    scalar (AS_ADD,AS_DELETE)               act:[AD]
 eaccession: Fragment_ID
 iaccession: IntFrag_ID                             acc:(%d,%d)
 variant of action:
  AS_ADD:
   record
    type:  scalar (AS_READ,AS_GUIDE)                typ:[RG]
    source:     "description of data source"        src:↵(%[^\n]↵)*.
    entry_time: time_t                              etm:%d
    clear_rng:  SeqInterval                         clr:%d,%d
    screened:   sorted list of ScreenMatch          scn:↵(<SMA-record>↵)*.
   end
 end
end                                                 }
```

### 2.2.2 Overlap Records

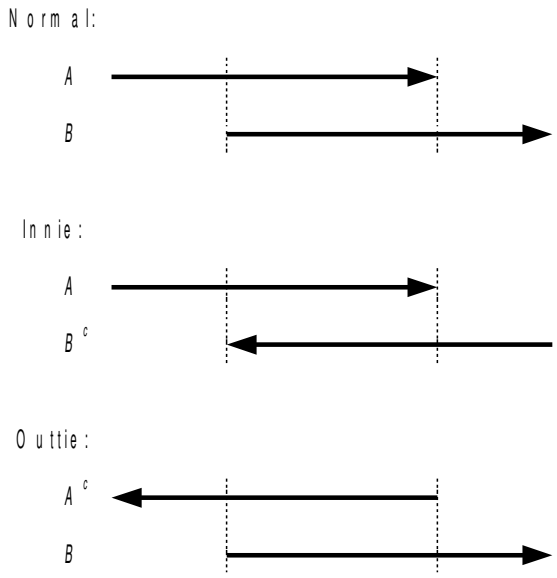
Overlaps will be represented by the following data structure:

```
OverlapMesg:                                            {OVL
record
 aifrag:       Int_Frag_ID                              afr:%d
 bifrag:       Int_Frag_ID                              bfr:%d
 orientation:
   scalar (AS_NORMAL,AS_INNIE,AS_OUTTIE)                ori:[NIO]
 overlap_type:
  scalar (AS_DOVETAIL,AS_CONTAINMENT,AS_SUPERREPEAT)    olt:[DCS]
 a_hang:       int                                      ahg:%d
 b_hang:       int                                      bhg:%d
 quality:      float                                    qua:%f
 min_offset:   int                                      mno:%d
 max_offset:   int                                      mxo:%d
 polymorph_ct: int                                      pct:%d
 delta:        string (int)                             del:↵((%d)*↵)*.
end                                                     }
```

- The `aifrag` and `bifrag` values are the unique ID's of the two fragments whose overlap is reported in this record.
- The `orientation` value indicates the direction in which the fragments overlap.

In all cases the *A* fragment begins at the same position or to the left of the *B* fragment. The possibilities are:

- `AS_NORMAL` — A suffix of *A* matches a prefix of *B*.
- `AS_INNIE` — A suffix of *A* matches a suffix of $B^C$.
- `AS_OUTTIE` — A prefix of $A^C$ matches a prefix of *B*.




- The `overlap_type` indicates which general type of overlap this is. Dovetail and containment overlaps are the normal cases. The distinction between them is redundant—it can be inferred from `orientation`, `a_hang`, `b_hang` (described below), and the fragment lengths.
  The type `AS_SUPERREPEAT` is a special case that indicates that *A* contains a subregion that occurs too frequently in the genome for reporting all overlaps. In this case, the `bifrag` value, if present, would indicate the longest such overlap detected in this batch of overlap comparisons.
  [Note to self: Need to be careful that the reported overlap is in fact one of the microsat-repeats. There may be a longer overlap, extending beyond the superrepeat region, that should be reported as a normal overlap.]
  If no `bifrag` value is given for a superrepeat, then the `orientation` must be either `AS_NORMAL` or `AS_OUTTIE` and the general type is automatically `AS_DOVETAIL`. At this point it is not settled how these will be detected. It could be done purely by k-mer frequency; by counting the number of matches to other fragments and stopping after some threshold is exceeded; or by the UR detector, in which case it might be ignored by the Overlap Detector.
- `a_hang` and `b_hang` indicate where the start and end of the alignment of the matching region of fragment *B* is relative to fragment *A*. `a_hang` is always non-negative, and is the number of positions (bases) that the *A* fragment extends out in front of the leftmost base (the first base in the clear region) in *B* (or $B^C$). Similarly, `b_hang` is the number of bases that the *B* fragment extends out past the end of the *A* fragment. For a dovetail overlap the value of `b_hang` will be

positive. For a containment overlap, `b_hang` will be non-positive.

- `delta` is a 0-terminated sequence of integers that describe the remainder of the alignment between the matching regions in *A* and *B*. Specifically, these integers indicate the indel positions in the alignment relative to the previous indel. A negative value indicates a missing entry in fragment *A*; a positive value denotes a missing base in *B*. For example, the following alignment:

  A: `cgt--tattacgtcg`
  B: `cgtgatat--cgacg`

  would be expressed by the delta sequence:
  `-4, -1, +4, +1, 0`
  The absolute value of each delta value represents the number of positions forward from the current position that the next indel occurs.
  The initial delta position is the start of the overlap region. *E.g.*, in the following overlap alignment:

  A: `cattagcggtatcgacgacgacga ...`
  B: `cgct-tcgacgacgacga ...`

  `a_hang` is 6 and the first delta value is 5.
  Delta values will be restricted to range from –128 to 127 so that they may be stored in a single byte. Two special values will be used:

  - -127 will be used when the offset to the next indel is more than 126 bases. The effect of the -127 is to shift the alignment position forward by 126 bases. Thus, an alignment that consists of a single insert in *B* at position 300 would be expressed by the delta sequence:
    `-127, -127, +48, 0`
  - -128 will be used for alignments with gaps that appear to represent small polymorphisms. This value will indicate the presence of a larger-than-usual gap. The signed integer following the -128 will denote the length of that gap. Note that this value is in addition to the indel represented by the integer preceding the -128. For example, the following alignment:

    A: `cggt---------aggaacg ...`
    B: `cgctacgacgacgacga-cg ...`

    would be expressed by the delta sequence:
    `-5, -128, -8, +5, 0`
    and similarly, the alignment:

    A: `cggtacgacgacgacgaacg ...`
    B: `cgct---------acg--cg ...`

    could be expressed by the delta sequence:
    `+5, -128, +8, +4, +1, 0`
    Note that the sign of the value after the –128 will always be the same as that of the value in front of the –128.
    The rules used to determine exactly what constitutes a “small polymorphism”

are discussed further below. It is likely that they will evolve with experience. Note that small polymorphisms that are not indels have no special delta code. Nonetheless, they are included in the value of `polymorph_ct` described below.

- `quality` is an indication of how likely it is that the matching regions in fact represent samples from identical regions of the genome. Details are yet to be determined. The alternative hypothesis is that the matching regions were sampled from different regions that were, say, 95% similar. In the current version of the program, this value is simply the error fraction in the overlap region, so that a *small* value indicates a *close* (*i.e.*, high-quality) match.
- `min_offset` and `max_offset` indicate whether there are alternative alignments that result from the matching region being (nearly) periodic. `min_offset` is the smallest possible value that the `a_hang` field could have and still produce a valid overlap with fragment *B*. Similarly, `max_offset` is the largest value of `a_hang` that could represent a valid alignment. If there are no alternative overlaps, then the values of `a_hang`, `min_offset`, and `max_offset` will all be the same.
  For periodic dovetail overlaps that contain at least two full periods of the overlap sequence, the value of `max_offset` will be greater than the length of (the clear region of) fragment *A*. If there are not two full periods in the overlap, then `max_offset` will only report the alternative alignment if that alignment meets the usual criteria for an overlap.
  For containment overlaps in which fragment *B* is completely periodic, the values of `min_offset` and `max_offset` will indicate the leftmost and rightmost alignments of *B* within *A*, respectively.
  If fragments *A* and *B* are both completely periodic (and either of them contains two full periods), `min_offset` will have a negative value and `max_offset` will be greater than the length of (the clear region of) fragment *A*.
  The alignment reported by the `a_hang` field is the one with the best `quality` value. In case of ties, the smallest positive `a_hang` value (representing the maximum overlap) will be used.
  Assume, for example, that in the following case the alignment shown is the best possible based on quality values, and that its `offset` value is 300:

  A: … `aaaacgtcgtcgtcgtcgtcgtcgtcgt`
  B: `cgtcgtcgccgtcgtcgtcgtcgtcgtcgtaacgaa` …

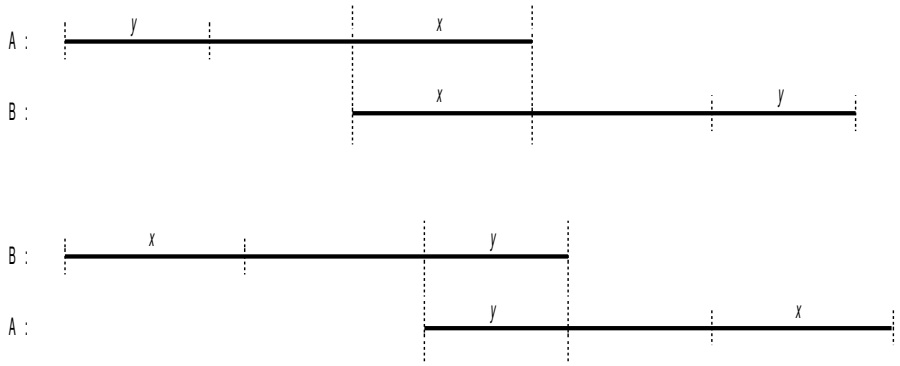
  Then the value of `min_offset` would be 297, reflecting the maximum overlap, and the value of `max_offset` would be 324, reflecting that the overlap region is purely periodic.
  The shift values will only be computed if the overlap region contains at least 2 complete copies of the repeat or alternative exact *k*-mer matches.
  Note that the match region of branch overlaps also can be periodic. In such cases the branch points and corresponding match ends will be reported at their extremal

positions.

A: … agtaggacgtcgtcgtcgtcgtcgtcgtaaagggttt …
B: … gcccaagcgtcgtcgtcgtcgtcgttacgaa …

- The value of `polymorph_ct` indicates the number of (presumed) small polymorphisms contained in this alignment. These are relatively short mismatch regions that are long enough to cause failure of the standard match-quality test, but are bounded by regions that strongly suggest the match is not random. They also can be viewed as small gaps between two branch overlaps. Most commonly, (perhaps exclusively) they should be extra copies of microsatellite repeats.

- Note that it is possible to output more than one overlap record for the same pair of fragments, as in the following case:




Since these overlaps are not caused by periodicity, they are not indicated by values of `min_offset` or `max_offset`. For a given pair of fragments, there will be no more than 2 overlaps reported.

### 2.2.3 Immutable Facts

- `a_hang` will always be non-negative.
- `overlap_type` is `AS_CONTAINMENT` iff `b_hang ≤ 0`.
- If `overlap_type` is `AS_DOVETAIL` then `a_hang > 0` and `b_hang > 0`.
- Overlaps never refer to the reverse-complement of ***both*** fragments.
- If `a_hang = b_hang = 0` then `ia_frag < ib_frag` (internal ID's). Not yet implemented.
- For both `AS_INNIE` and `AS_OUTTIE` dovetail overlaps, `ia_frag < ib_frag` (internal ID's). Not yet implemented.
- All internal fragment ID's are positive integers less than $2^{31}$.

### 2.2.4 Branch-Point Records [Obsolete]

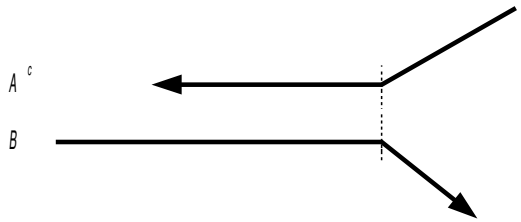
Branch points are no longer output by the overlapper.
Branch points will be output using the following simple data structure:

```
BranchMesg:                                                    {BRC
record
 action:     scalar (AS_ADD,AS_DELETE,AS_UPDATE)               act:[ADU]
 ifrag:      Int_Frag_ID                                       frg:%d
 pre_br:     int                                               pbr:%d
 suf_br:     int                                               sbr:%d
 pre_end:    int                                               pen:%d
 suf_end:    int                                               sen:%d
end                                                            }
```

- `action` indicates whether this is:
  1. The first report of branch-point information for this fragment (`AS_ADD`).
  2. The removal of all previously reported branch information for this fragment (`AS_DELETE`). This should occur only because of the deletion of the (one and only fragment) that induced the branch point on this fragment. In this case the values of `pre_br`, `suf_br`, `pre_end`, and `suf_end` all will be zero.
  3. A change in previously reported branch-point information for this fragment (`AS_UPDATE`). This would commonly occur when a new fragment implied a larger repeat region on this fragment. All 4 fields are included in the message, even if only one value has changed.
     Suppose fragment *a* is the only fragment that induces a prefix branch point on fragment *b*, but that other fragments induce suffix branch points on *b*. If *a* is deleted, then an `AS_UPDATE` message will be sent for fragment *b* with zeroes in the `pre_br` and `pre_end` fields is deleted, but the current extremal values in the `sub_br` and `suf_end` fields. The symmetric case applies when the last suffix branch is deleted, but prefix branches remain.

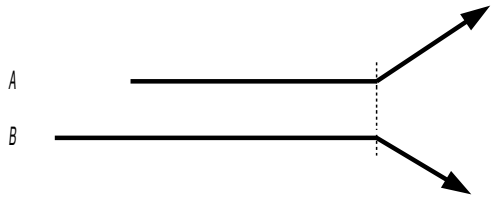
  See below for a further discussion of what branch-point information will be maintained in the Fragment Store.
- `ifrag` is the internal ID number of the fragment containing the branch point information.
- `pre_br` is the location in this fragment at which the earliest (*i.e.*, leftmost or closest to the 5' end) prefix branch point detected so far has occurred. A *prefix branch point* is one that indicates a boundary between differing sequence on its 5' side and matching sequence on its 3' side. This position is relative to the fragment in which it is reported. The orientation of the other fragment does not matter. Thus, in the following case (where arrows point toward the 3' end of the sequence):

  the indicated branch point is a prefix branch point in both *A* and in *B*, whereas in the following case:

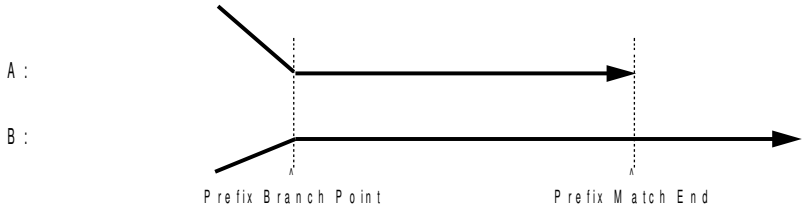



the indicated branch is a prefix branch point in *A* and a suffix branch point (see below) in *B*.
The specific location value is determined by maximizing a transformed distance function along the path of the best alignment, as described below. It will be the position between the "last" base that matches and the next base, which must be a mismatch.

- `suf-br` is the location in this fragment at which the latest (*i.e.*, rightmost or closest to the 3' end) suffix branch point detected so far has occurred. A *suffix branch point* is one that indicates a boundary between matching sequence on its 5' side and differing sequence on its 3' side. The following:




is a suffix branch point in both *A* and *B*.
- `pre_end` is the maximum position (*i.e.*, closest to the 3' end) to which the end of the matching region of any prefix branch overlap detected so far extends.




Note that the values of `pre_end` and `pre_`br need not come from the same branch overlap. Each is the separate extremal value from the set of all prefix branch overlaps.
- `suf_end` is the minimum position (*i.e.*, closest to the 5' end) to which the end of the matching region of any suffix branch overlap detected so far extends. As before, `suff_end` and `suf_`br need not come from the same branch overlap.
- All reported values are relative to the clear range of the fragment.

## 3. Algorithm Specifications

*Assumption:* All overlap calculations will be done using only the clear region of fragments.

### 3.1.Detecting Match Regions

*Definition:* An *overlap* between two fragments will consist of subsequences of the fragments, both at least MIN_OLAP_LEN bases long, between which the number of errors (character inserts, deletes and mismatches) is less than MAX_ERROR_PERCENT of the shorter length of the two subsequences.

Note that this does not use quality values. The quality values are used to determine the best alignment. Overlaps will be found by finding exact matches of length *k*, and then checking whether they can be extended to the necessary length within the designated error bound. The value of *k* will constrain the values of MIN_OLAP_LEN and MAX_ERROR_PERCENT. A value of $k = 13$ supports MIN_OLAP_LEN = 40 and MAX_ERROR_PERCENT = 0.05 under this definition. This value of *k* is probably too small for efficient computation. Using a larger value of *k* may result in some overlaps being missed.

The algorithm will process fragments in batches. Each batch will consist of two streams of fragments: *New* and *Old*. The output of the algorithm will be a representation of all overlaps between any fragment in *New*, and a fragment (or its reverse complement) from *Old*. Note that reverse-complements of fragments from *New* are not considered, and that no attempt is made to find overlaps between 2 fragments in *New*, or between 2 fragments in *Old*. There is no requirement that *New* and *Old* be different streams, however. If they are the same stream, then all overlaps between different fragments in that stream will be produced. No overlap will be reported between a fragment and itself (either forward or reverse-complement).

Note that overlaps will be checked between fragments and their reverse-complement clone mates as a check against short clones. Overlaps also will be checked between each fragment and its forward clone mate.

The algorithm begins by reading all the fragments from *New* and creating an index of all the *k*-mers contained in them. The current implementation of the index is a hash table. Then fragments are read from *Old*, and processed one at a time. Let *u* be the fragment from *Old* being processed. Each *k*-mer in *u* is searched for in the index, and all matches with fragments from *New* are recorded. Matches are partitioned into groups representing identical matches longer than *k*, since an exact match of length $m > k$ consists of $1 + m - k$ overlapping *k*-mer matches. Each group is processed separately.

When processing a batch of fragments against itself, to prevent duplicate reporting of overlaps, only overlaps between fragments where the *New* fragment ID is greater than the *Old* fragment ID will be considered.

From the resulting maximal, exact matches the longest region *R* is selected. Starting at *R*, two edit-distance calculations are performed: one to the left and one to the right to determine how far the match region can be extended within the MAX_ERROR_PERCENT error constraint. If the resulting match is sufficiently long and extends to the end of at least one of the fragments, it is analyzed to determine its type and alignment. If it is a complete overlap, it is output. If it is a branch overlap, its information is used to update the branch-point information for each of the fragments.

The other groups representing maximal exact matches between *u* and *v* are then analyzed. If they are contained within a previous match, they are ignored. Otherwise,

they are processed in the same fashion. Note that it is possible to report more than one overlap between a single pair of fragments *u* and *v*, as mentioned earlier.

### 3.2.Extending the Exact-Match Region

To extend the exact-match region *R* above, a simple edit-distance calculation (I use the *O* (*Errors* × *Length*) algorithm of Ukkonnen/Myers/Landau-Vishkin with a "band" eliminating any alignments that are unlikely to extend to the end) can be done to compute an optimal alignment and determine how far the match can be extended without exceeding the `MAX_ERROR_PERCENT` error constraint. If the match extends to the end in both directions, the alignment and quality are computed and the overlap is output.

If the match ***almost*** makes it to the end, a few bases of fudging are allowed. The maximum number of such bases is `OLAP_SLUSH`.

If the match does not extend to the end of either fragment, then the branch point needs to be determined. We can do this by scoring the above alignment using a transformed distance measure δ'. The peak score along the alignment using this distance will mark the end of the overlap match region.

Specifically, for two characters *a* and *b* we define δ' (*a*, *b*) = *x* if *a* = *b* and δ' (*a*, *b*) = *y* if *a* ≠ *b*. We wish to choose the values of *x* and *y* so that along the alignment path, if the sequences match at the expected rate of, say, 4%, then the slope of the graph of distance versus path length is +1. And if the sequences are random, which corresponds to a match rate of, say, 66%, then the slope of distance versus path length is –1. [Using these numbers I get *x* = 1.08 and *y* = -4.72.] The end of the match region, *i.e.*, the branch point, will be the position along the alignment path at which the cumulative δ' score is maximum.

A branch overlap is discarded unless the length of the non-matching part of both fragments is sufficiently long (`MIN_BRANCH_END_DIST`).

### 3.3.Recording and Reporting Branch Points [Obsolete]

The overlapper no longer outputs branch-point information.

We expect huge numbers of branch overlaps because of the frequency of repeat regions in genomic data. It is therefore impractical to report each one of these as a branch point. Instead we propose to maintain for each fragment just information about the *extremal* branch points. Specifically, we shall record the leftmost prefix branch point, the rightmost suffix branch point, the rightmost prefix match end, and the leftmost suffix match end.

Because of sequencing error, the exact position of each branch point is uncertain. Therefore we shall keep with each branch point a small array of counters representing a histogram of the number of times another fragment induced a branch point at or near the extremal value. These counters can be regarded as "votes" by other fragments as to where the branch point should be.

Thus the `Branch_Info` structure we keep for each fragment will consist of a separate `Position` and a `Votes` array pair for the leftmost prefix branch point and for the rightmost suffix branch point. `Position` will be the coordinate in the sequence where the extreme branch occurred, calculated as described in the previous subsection. `Votes [i]` will be the number of other fragments that voted for the branch point at

`Position + i` (for prefix branch points) or `Position - i` (for suffix branch points).

The branch-point information stored with each fragment will be updated with each batch of fragments processed by the overlapper. From a single batch, at most one branch-point message per fragment can be emitted. If the position of the branch point is unchanged, no message is emitted, even if the `Votes` values have changed.

[Note to self: Need to worry about synchronization issues in updating `Votes` when running in parallel.]

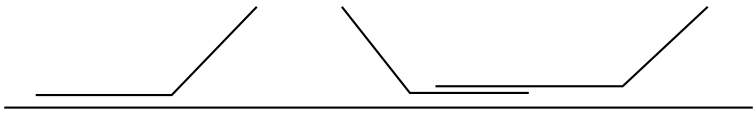
Updating the branch information for an additional vote is straightforward. If the new position is within the extremal recorded, the appropriate vote count is incremented (if it exists in the array—it may be off the end); otherwise, the extreme position is modified, its vote set to one, the vote counters shifted the appropriate amount, and a branch update message emitted.

The match-end values are simply the max (for prefix) and min (for suffix) values encountered so far.

When a fragment is deleted, its overlaps with all other fragments will be recomputed, and the appropriate branch votes decremented. If the vote count at the extremal position goes to zero there are 2 cases: If another vote count is non-zero, the votes can be shifted and a branch update message emitted. If *all* vote counts are zero, then overlaps for this fragment will need to be re-computed to re-determine its branch points if any. If it has none, then the branch delete message is emitted. This is the *only* case in which the branch delete message is sent. The branch points on a deleted fragment itself are assumed to be deleted without sending a redundant message to that effect.

The effect of a delete fragment on match ends is yet to be determined. It could be recomputed, as described above. Or in the case where there are sufficient votes for the branch-point, it may simply be kept even though it might be erroneous.

The branch-point information conveys an estimate on what regions within a fragment have evidence of being repeat regions in the genome. Specifically, the union of intervals [`pre_br`, `pre_end`] ∪ [`suf_end`, `suf_br`] is presumed to be in repeats. Note that this overestimates what is repeats in the case of the following overlaps:



where the central region is assumed to be repeat, even though it very well may be unique.

## 3.4.Analyzing the Match Region

### 3.4.1 Current Status

The overlap currently reported is the one with the minimum error rate, which is defined as the number of errors divided by the shorter match region of the two fragments. (Note that because of inserts and deletes, the length of the portion of the fragment that aligns with the other fragment may be different in the two fragments. It is also possible that the best match might not be found in the presence of microsatellite repeats with small

variation.) Quality values are not used in counting errors.

Overlaps are also checked to see if they contain a subregion with a disproportionately high incidence of mismatches. This check is implemented by sliding a fixed-length window along the alignment of the overlap region and counting the number of errors, taking into account quality values as follows: Let $q$ be the lower quality value of two aligned, mismatching bases (or the quality value of an inserted or deleted base). If $q$ is less than a threshold value $T$, then we count it as $q/T$ of an error; otherwise, it counts as a full error. If the error count exceeds some specified limit in any window of length $W$ in the overlap region, the overlap is rejected and not output. Currently, the overlap module is set to reject any overlap with either 8 or more errors in a window of length 50, or 12 or more errors in a window of length 100.

### 3.4.2 Other Possibilities

The alignment reported will be the one with maximum likelihood based on quality values. Exactly what this means is yet to be determined. One possibility is to create a distance function that is the negative log of the probability that a pair of characters matches, based on their quality values. The best alignment is the one that minimizes the sum of these distances over the alignment.

The definition of the overall quality-of-match value is not yet determined. The preceding distance measurement could be used, normalized by dividing by the length of the match region to obtain an average quality-per-base value. An alternative is to develop a maximum likelihood score to specifically determine if the match is more likely to be from sequencing error or from a chance match with a similar region. Such a score could be derived empirically from simulations.

Detecting periodicities in the exact-match portions of the overlap match region is easy when there are multiple matches of the *k*-mers. For short matching regions additional analysis may be necessary. This can be based on periodicities within the *k*-mers that could be built into the *k*-mer index.

Long, inexact periodicities that contain neither a separate exact *k*-mer match nor a periodic *k*-mer are harder to detect. [I think they are possible, but they should be exceedingly rare. I don't know how much effort should be expended to try to detect them or what the cost might be.]

## 3.5.Finding SuperRepeats

SuperRepeats are overlap match regions that occur too frequently to be worth reporting. Because the old fragments are processed in batches, it's possible [likely, I think] that such regions may not occur too frequently within a given batch, but collectively in all the batches they may be too frequent. In that case, the post-processing step that adds the output of the overlap detector to the overlap database could detect the superabundance and make a special entry in the overlap database. Note that this implies changes in the overlap database that might modify the overlap graph and/or unitig graph. The same phenomenon occurs if a SuperRepeat is detected in a single batch that had not been detected in prior batches.

The ubiquitous-repeat tagger (URT) that processes fragments before they reach the

overlap module will detect regions within fragments that should not be included in the *k*-mer index from which overlaps are computed. No *k*-mer more than half contained in such a region will be put in the *k*-mer hash table.

In addition, the hash table will record the number of hits against it from *Old* fragments. (Sub)sequences with an excessive number of such hits can be flagged and/or fed back to the URT.

Another possibility is to group the old data into batches so that SuperRepeats occur together. Such a partitioning may be hard given fragments that contain different SuperRepeats.

### 3.6. Finding Small Polymorphisms

Small polymorphisms will be detected when there are two or more regions that meet the length and match-quality criteria, but which do not extend to meet each other. Whether additional criteria should be included is not clear. Certainly a region that represents 1 or more extra copies of a small repeat will be included if it is not too long. Polymorphisms that are not variations in the number of small tandem repeats will be allowed only if their length is not more than `OLAP_SLUSH`.

Non-matching regions at the end of either fragment will not be reported as polymorphic overlaps—they will be reported as branch points.

### 3.7. File Aspects

Fragments are processed in batches. The size of the batch is determined by the number of fragments that can be indexed in memory. Other than finding SuperRepeats, maybe, the grouping into batches is arbitrary.

The Overlap Index is maintained by the modules that receive the outputs of the Overlap Detector Module.

## 4. LSF Overlapper

The LSF version of the overlapper is designed to handle very large overlap tasks. It works by first invoking a script-generating program that analyzes the size of the fragment store and the size of the new fragments to be added to it. It then generates a script of LSF commands to launch a series of overlapper jobs that can be run anywhere on the compute farm. These jobs collectively do the entire overlap computation. As a final step their output files can be concatenated to be passed along to the unitigger module.

The general strategy will be:

1. Insert the new fragments into the fragstore. This will allow all subsequent jobs to read the store simultaneously without any special synchronization. It also permits random access to any range of fragments.
2. Submit a collection of jobs with parameters specifying two ranges of fragments: the fragments to put into the hash table (analogous to “new” fragments in a normal run); and the fragments to stream against the hash table (analogous to “old” fragments). These jobs are completely independent and each writes its own .ovl file.
3. When all the jobs in the preceding step have completed, their output files can be concatenated.

## AUTHORS

Art. Delcher:

Revised: 8 Jan 99
Added Sections 2–7 and updated message formats.

Revised: 29 Jan 99
Moved sections to `/doc/Assembler/Programs/OVLManual.rtf`, made changes caused by the gatekeeper, changed branch-point message and added description of maintaining branch-point info.

Revised: 25 Feb 99
Revised branch-point information, and other small revisions to reflect current overlapper.

Revised: 1 Mar 99
Fixed 3-codes in branch message.

Revised: 30 Nov 99
Marked branch-point sections as obsolete and added section on LSF overlapper.

Revised: 8 Feb 2000
Small change to LSF section.