Module: Intersection Analysis (LIA)

Summary:

LIA (**L**ocus **I**ntersection **A**nalysis) does intersection analysis for sets of genomic loci. It performs comparisons among coordinate-based data in a high throughput manner, identifying shared or common regions. It allows for any number of sets of loci to be compared, each set can contain any number of loci, and loci can overlap within a set. A variable number of nucleotides can be defined for either minimum required overlap, or maximum allowed gap between loci. This minimum overlap or maximum gap can be set as either a fixed number, or a percentage. Also any set can be defined as a *negative* set, meaning it should not be in common with the others. Additionally a 'bridging' criterion is allowed, where a locus can span 2 other loci and *bridge* the intervening region.

Algorithm:

The LIA algorithm is rooted in simple set intersection. However, the data and comparable conditions hold some additional complexity. Each group of loci is a set which can intersect with other sets. But each set member (each locus) is not a discrete unit that can be defined as a member of multiple sets. In fact, each locus is itself a set - of nucleotides - and the nucleotides act as the discrete unit of comparison. Thus the requirement becomes the analysis of *sets of sets*.

There are caveats within the conditional comparisons as well. For instance, multiple loci within the same set are able to intersect with each other (e.g. isoforms of a gene). Also when comparing loci, the determination of a true/false intersecting condition is variable, given the user-defined parameters. This means that loci can share any number of nucleotides, or even none at all (allowing for a gap), and still be considered a true condition. Lastly a bridging criteria can be considered, which forces a simultaneous comparison among elements of 3 or more sets, allowing for more complex truth conditions.

To maximize efficiency, LIA applies an *ordered set and sweep* concept to move through the data. It works similarly to the Bentley-Ottmann [1] algorithm for finding the set of intersection points for a collection of line segments in 2-dimensional space. The Bentley-Ottmann algorithm enforces a linear order on the line segments, using their endpoint coordinates. It then proceeds by conceptually moving a "sweep line" across the coordinate plane, tracking all the lines/intersections as it encounters them. LIA works in a similar fashion. The loci within each input set are ordered based on their genomic coordinates. This allows LIA to organize each data set in a virtual linear model, and then "sweep" across them, minimizing the number of comparative permutations that must be generated. Due to the possibility of intersecting loci within a single set, there are a minimum number of iterative permutations that must be computed. However by utilizing the ordered nature of the data and hierarchical data structures, these permutations are isolated to many small scopes, and the resource requirement is minimal.

I - Nomenclature:

1) In LIA the loci are addressed in a linear order within their context, and directionality is implicit within the coordinates. It doesn't matter whether the biological directionality of the loci is 5' → 3', p → q, etc; and LIA does not need to make any assumption. However for reference purposes, the end of the context with the lowest number coordinates is referred to as the "low end", and the end of the context with the highest number coordinates is referred to as the "high end". Thus the locus closest to the low end is referred to as the ***"low-end locus"***. The next locus in order is the ***"next low-end locus"***. Etc.

2) Input data sets can be defined in two ways: they "should intersect" or they "should *not* intersect". Sets that should intersect are hereafter referred to as ***"positive sets"***, and sets that should not intersect are hereafter referred to as ***"negative sets"***.

II - Assumptions, data types and configuration:

A. Input data. LIA accepts data in the form of LocusSet objects (as defined in “Data Structures”).
B. Assumptions. LIA assumes that the input data shares the same genome context - such as species, build number, etc.; as well as the same coordinate system. Also LIA assumes that in each LocusSet, the loci of interest are those directly referenced by the LocusSet. If any Locus objects within the LocusSet contain a hierarchy (they have 'children' loci), the the hierarchy is not recursed and the child loci are ignored.
C. Bridging. Bridging is the condition in which 3 or more loci are being compared, and all loci only need to intersect with one other locus. For example: assume loci A, B, and C. A & B do not intersect, however if A & C do intersect and B & C do intersect, then C *bridges* A & B, and all three are considered to intersect.
D. Comparison type & comparison value. These parameters represent what the user defines as a true condition each time 2 loci are being compared. They are the same parameters as defined in “Data Structures” (Locus functionality), and indeed LIA utilizes this functionality directly as it proceeds through the analysis.
E. Non-Intersecting / Not in Common. The non-intersecting criteria allows for the negative condition to exist. Any data set that is loaded into LIA can be defined as not in common (negative), and should not intersect with the other data sets. For example, one could load Set1 (experimental results) to be intersecting with Set2 (phylogenetically conserved regions), and non-intersecting with Set3 (all genes). Thus the result would be conserved experimental loci that are intergenic.
F. Output. LIA produces 3 types of results:

- A subset of each original set, representing the loci which resulted in a positive condition.
- A set of regions, representing the aggregated loci which intersected with each other. These regions provide information about the union and intersection, as well as the original data points.
- A matrix representing the specific, unique groups of loci which intersected across all data sets.

III - Procedure:

See Fig 1

Each LocusSet given to LIA is prepped before the comparison algorithm begins. First the LocusSets are copied, in order to preserve the integrity of the original sets. Then the they are ordered, as described “Data Structures”. Lastly the LocusSets are compressed, again as described previously. This is done because the sweeping process can fail in certain instances when the data sets are not linear (ie: multiple loci overlap within the same set). For the compression process, the “wrapAll” parameter is used to tell the LocusSet to place *all* Locus objects into a 'region' container. This gives LIA a consistent data structure to work with.

The algorithm maintains a reference to one region from each set. The referenced regions are determined in an iterative fashion by virtually sweeping along the genome, and finding which set has the next low-end region. Once it is found, that set’s reference is changed to the newly discovered region, all referenced regions are evaluated for intersection, and the sweep continues.

See Fig 4 & 6.

For example in Figure 6, there are 3 sets of positive regions represented. The first 3 regions to be referenced and compared are A1-R, B1-R, and C1-R. After the comparison is made, each set is tested for existence of another region. Of the sets that do have another region (in this case they all do: A2-R, B2-R, and C2-R) those regions are examined. C2-R is determined to be the next low-end region, so Set C's current reference is changed to region C2-R, and the comparison is made among A1-R, B1-R and C2-R. Next, Set A's current reference is changed to region A2-R, and the comparison is made among A2-R, B1-R and C2-R. This procedure continues until all regions have been exhausted.

Each time regions are evaluated for intersection, the algorithm accounts for the user defined parameters - minimum overlap or maximum gap, and bridging. As stated previously, bridging allows for a true condition – a common region – among 3 or more loci, where 2 do not share a common region. For example in Figure 7A, when comparing A1, B1, and C1, B1 and C1 do not share a common region and the condition is considered negative without bridging - Fig 7B. However if bridging is allowed, locus A1 bridges B1 and C1, and the condition is considered positive - Fig 7C. The same phenomena will appear when the comparison is made among A4, B4, and C4. The comparison of these loci results in a negative condition without bridging, and a positive condition with bridging.

Each time referenced regions are determined to be positive for intersection, the algorithm branches. When this occurs, all permutations for the *individual loci* contained within the regions are examined. Each permutation of loci is evaluated for intersection, using the same criteria as the region comparisons. If a positive condition is found, then finally the "negative" data set condition is checked.

The negative LocusSets are treated similarly to the positive, except they are aggregated into a single LocusSet to reduce the conditional load. The negative LocusSet maintains references, which keep track of the current scope (the genomic coordinates) of the positive regions. This allows for 'checks' against negative regions to be kept to a minimum - only negative regions within the current scope need be checked. When positive intersecting regions are found, references to the negative regions are evaluated. If the currently referenced negative region is "before" the first positive region, then the reference is moved up to the next negative region. This process repeats until the current negative region is no longer before the first positive region - thus it is no longer out of scope. After the negative region reference has been updated, the permutations of *loci* within the positive regions are checked. When an intersection of loci is found, the final step is to compare these loci to the negative regions. The comparison starts at the currently referenced negative region (which is now in scope), and continues to compare against consecutive negative regions, but only until the negative regions are "after" the last positive region - thus out of scope.

As the iteration proceeds, each group of loci which have passed all the criteria are processed as positive results. This includes:

- Flagging all positive Locus objects from each LocusSet with a LIA-specific attribute. This allows LIA to quickly aggregate and return the *subset* of loci from each original LocusSet which passed the user's criteria. The return value is simply another LocusSet object.
- Assigning each positive group of loci to another data structure called a LocusNexus (*details forthcoming*). This a functional matrix which represents each specific Locus that intersects with each other specific Locus. This tells the user what exactly from Set A intersects with what exactly from Set B, etc., as illustrated by the following table using data from figure 7C:

Set A	***Set B***	***Set C***
A1	B1	C1
A1	B1	C2

Set A	*Set B*	*Set C*
A2	B1	C2
A4	B4	C4

- Assigning each positive Locus to an aggregate region. These regions are Locus objects which act as containers for positive loci. They perform 3 functions: They represent the largest total area occupied by all loci in the region – the *Union*. They hold all the original Locus objects which make up the region, tracking their annotation and the LocusSet they came from. Lastly they hold additional Locus objects representing the region(s) of *Intersection*. See figure 7C.

Any of the above result types can be requested from LIA after a single iteration of the algorithm. Each presents the results in a different manner, and which type the user chooses depends on the question being asked.

References:

[1] Jon Bentley & Thomas Ottmann, "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. Computers C-28, 643-647 (1979)

Potentially Novel Items:

1. Usage of an "ordered set and sweep" algorithm to quickly move through and evaluate all possible intersects of Locus data in an unlimited number of sets simultaneously.
2. Usage of a "linearizing" data structure to convert the data into a form usable by the ordered set and sweep algorithm.
3. Creation of a functional matrix (LocusNexus) to manage and manipulate resultant data in the form $(X \in A) \cap (Y \in B) \cap ...$
4. Creation of a aggregate data structure to manage groups of loci that share positive intersection criteria. This includes the area representing the Union, the area(s) representing the Intersection(s), and the original loci with accompanying annotation.

Figure 1

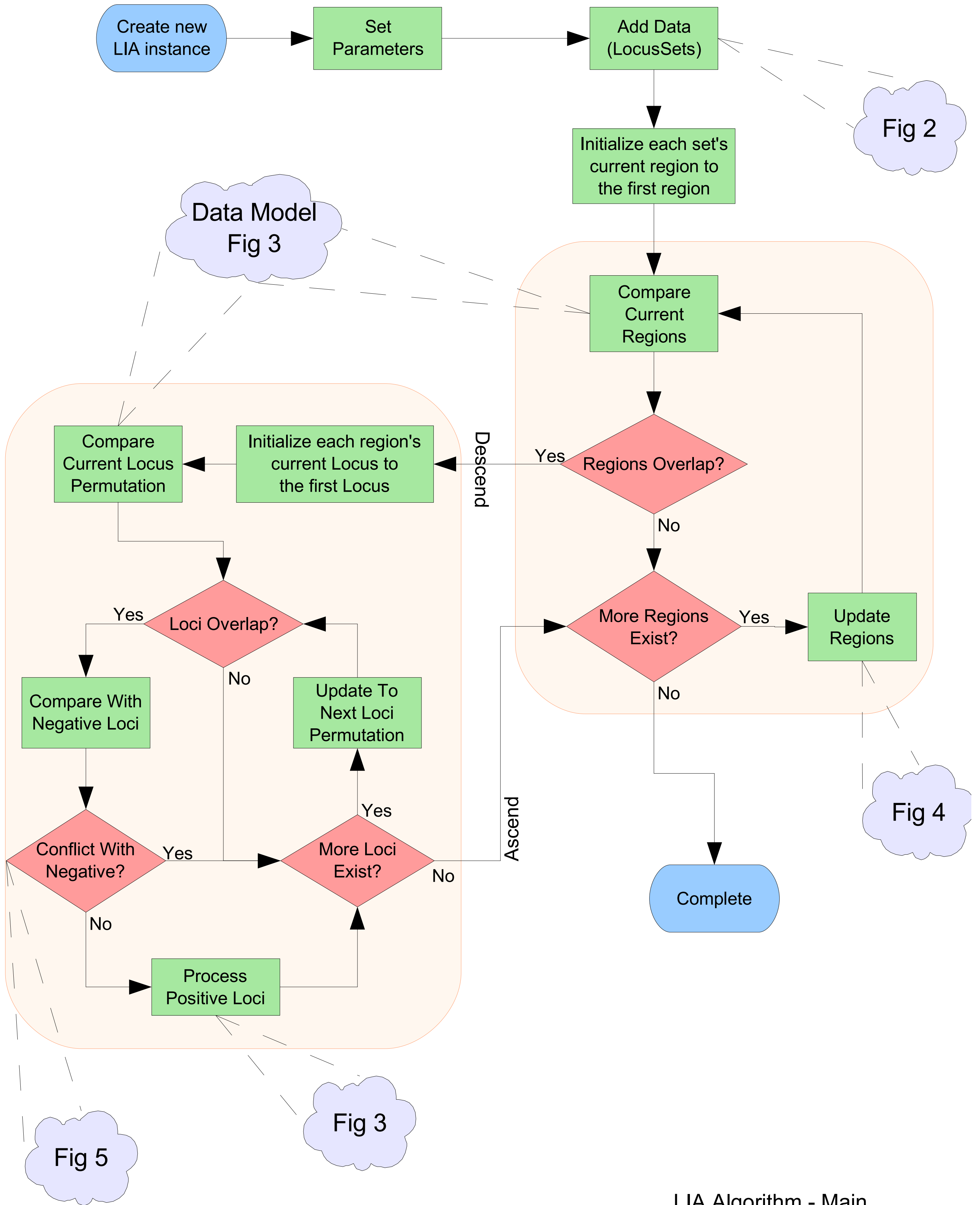
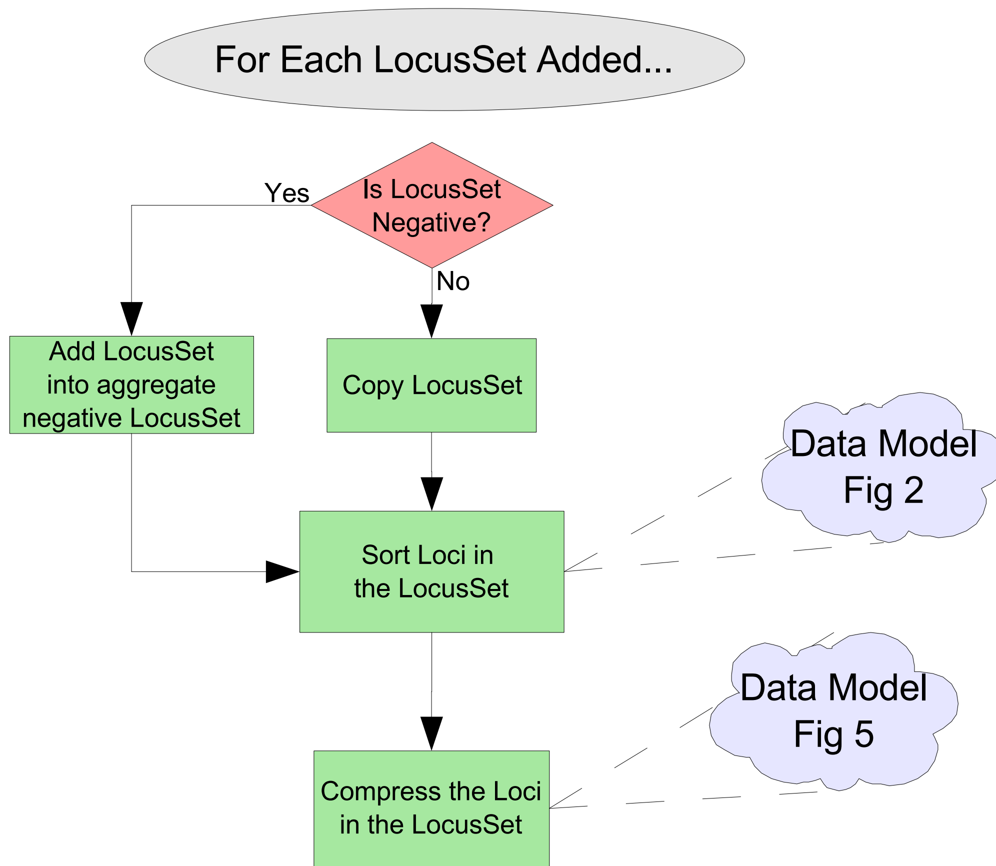
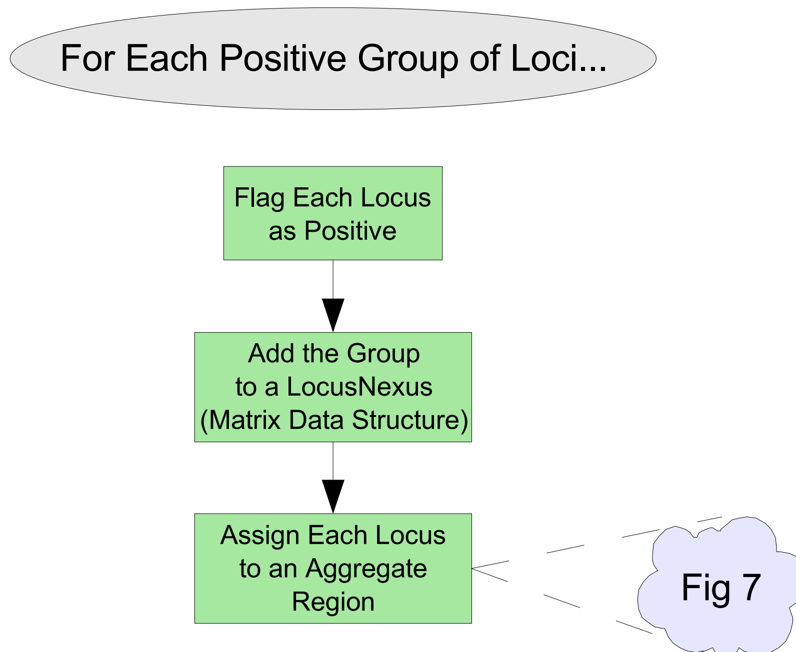


Figure 2



LIA – Add Data Set

Figure 3



LIA – Process Positive Loci

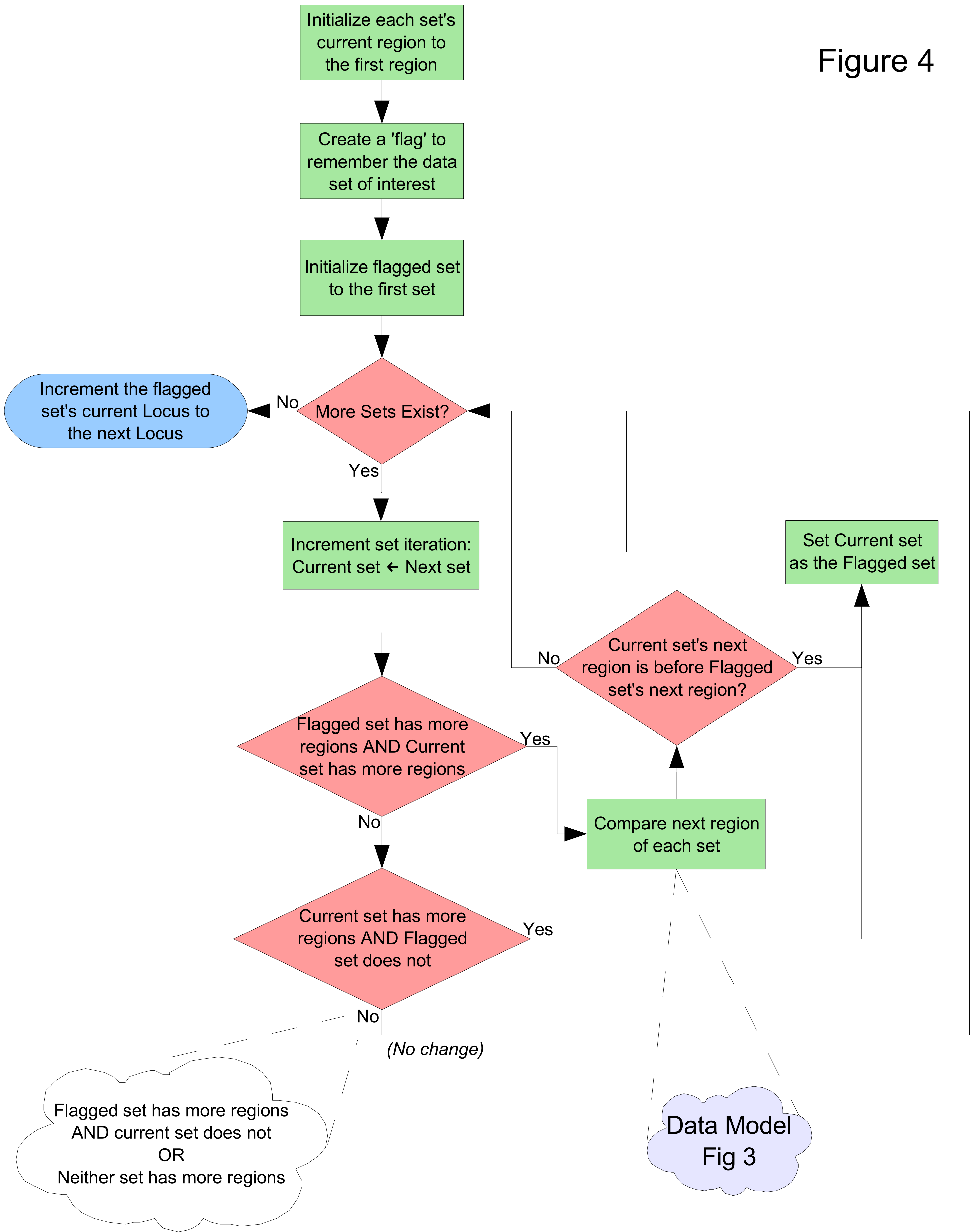


Figure 4
Initialize each set's current region to the first region
Create a 'flag' to remember the data set of interest
Initialize flagged set to the first set
Increment the flagged set's current Locus to the next Locus
No
More Sets Exist?
Yes
Increment set iteration: Current set ← Next set
Set Current set as the Flagged set
No
Current set's next region is before Flagged set's next region?
Yes
Flagged set has more regions AND Current set has more regions
Yes
No
Compare next region of each set
Current set has more regions AND Flagged set does not
Yes
No
(No change)
Flagged set has more regions AND current set does not OR Neither set has more regions
Data Model Fig 3
LIA – Update Regions

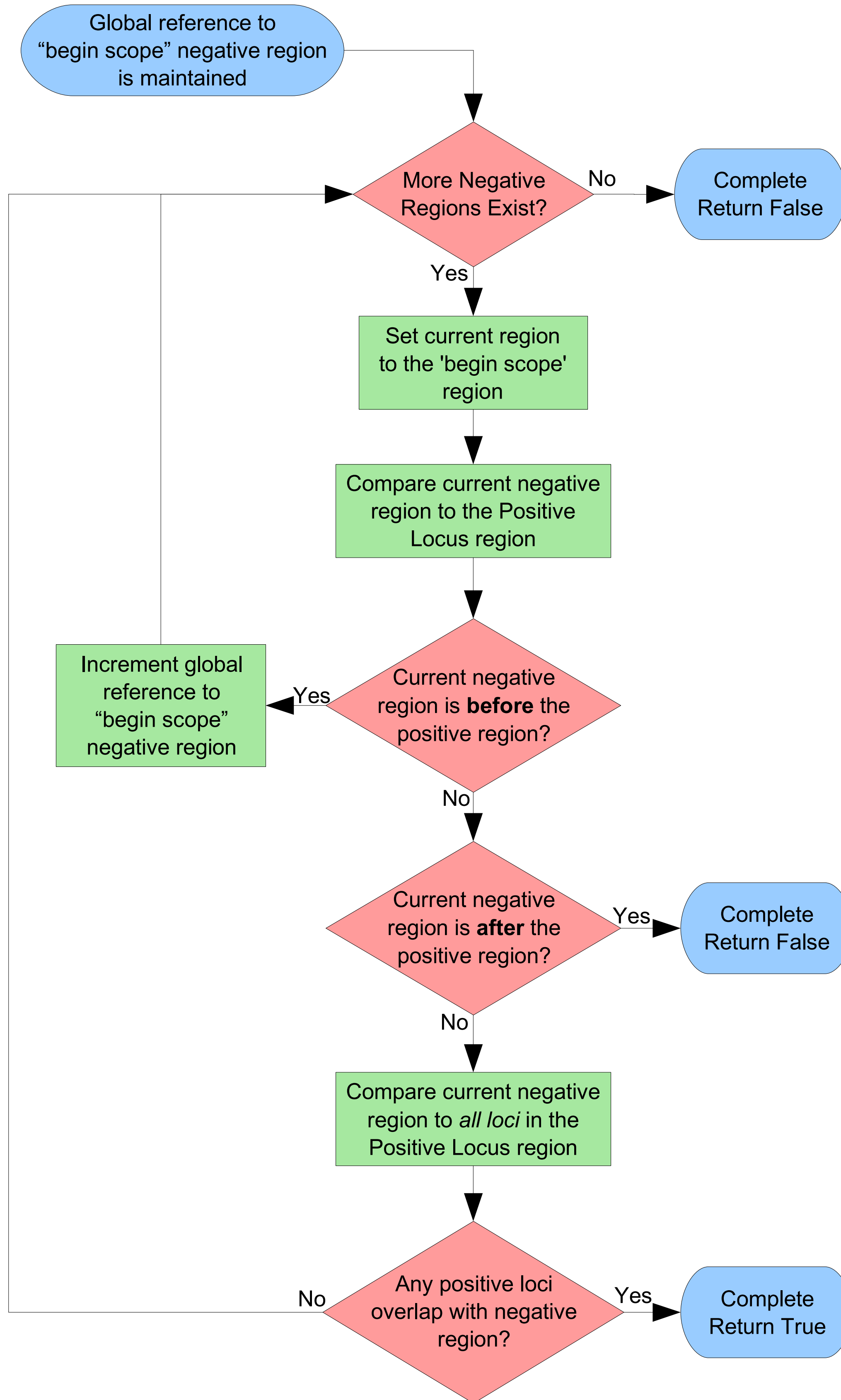
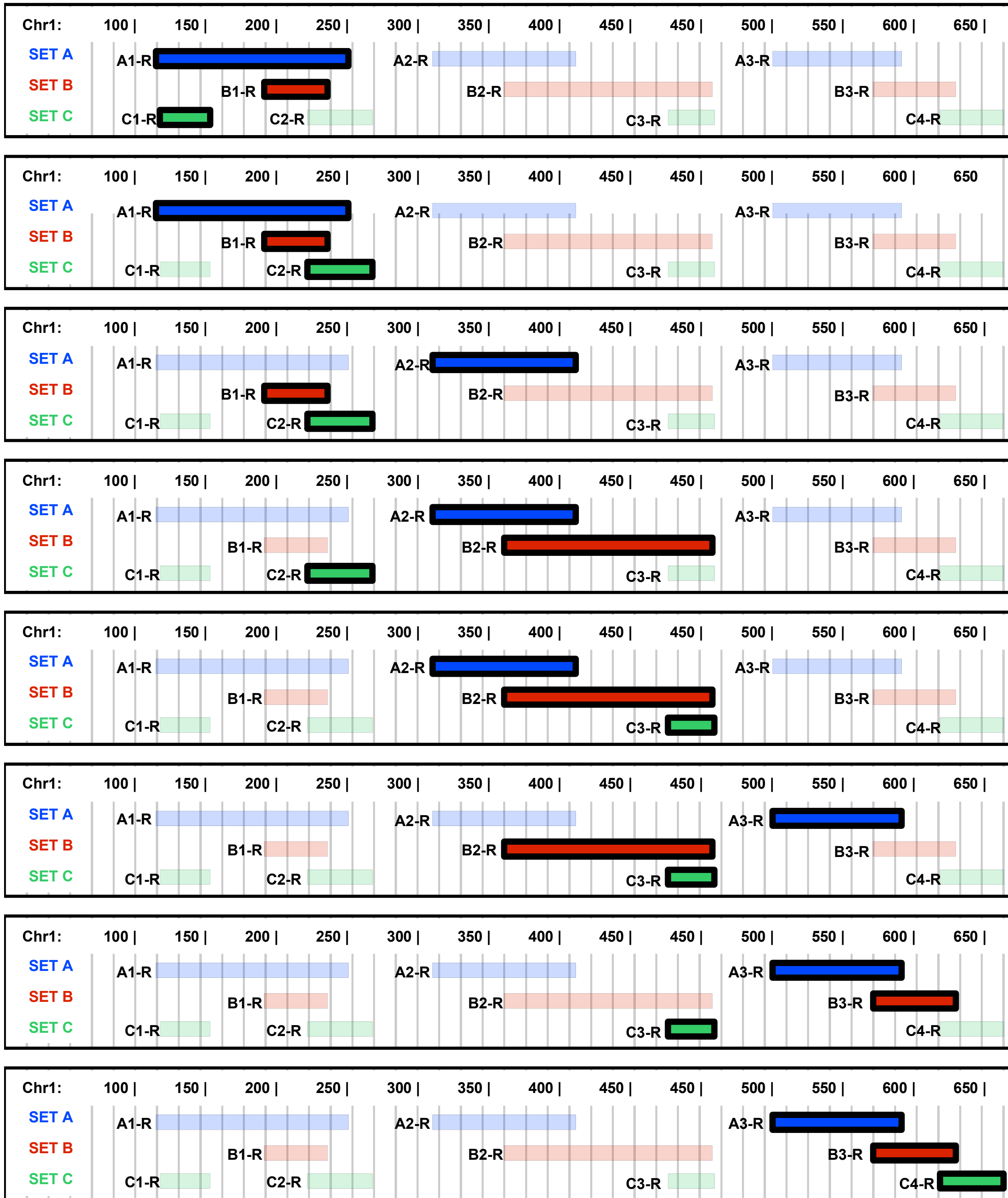


Figure 5

LIA – Conflict with Negatives

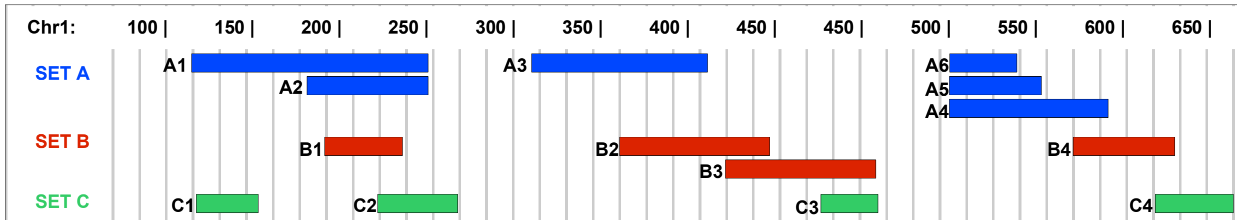
Figure 6



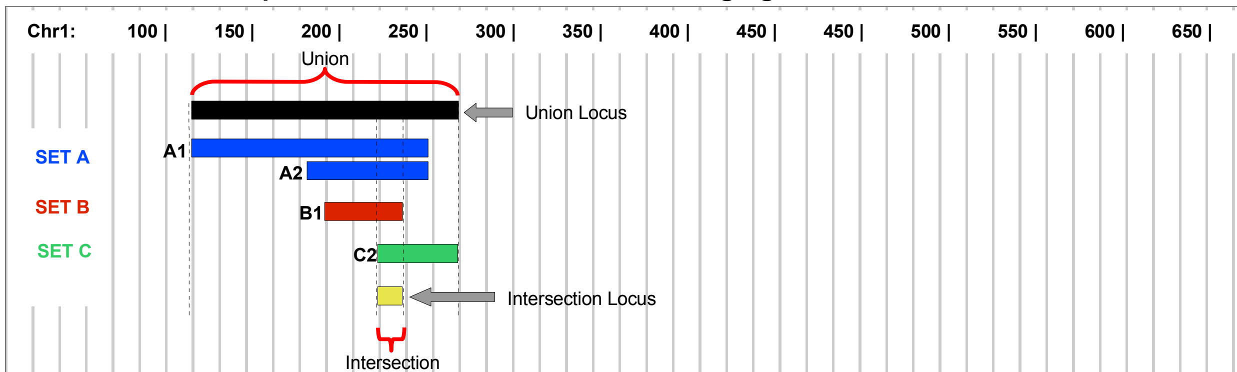
The sequence of figures from top to bottom shows the progression of loci that are examined as LIA “sweeps” across the genome. A reference to one region from each set is maintained, and after each group is examined, whichever set has the next “low-end” Locus has it's reference moved up. The new group is then examined, and process continues.

Figure 7

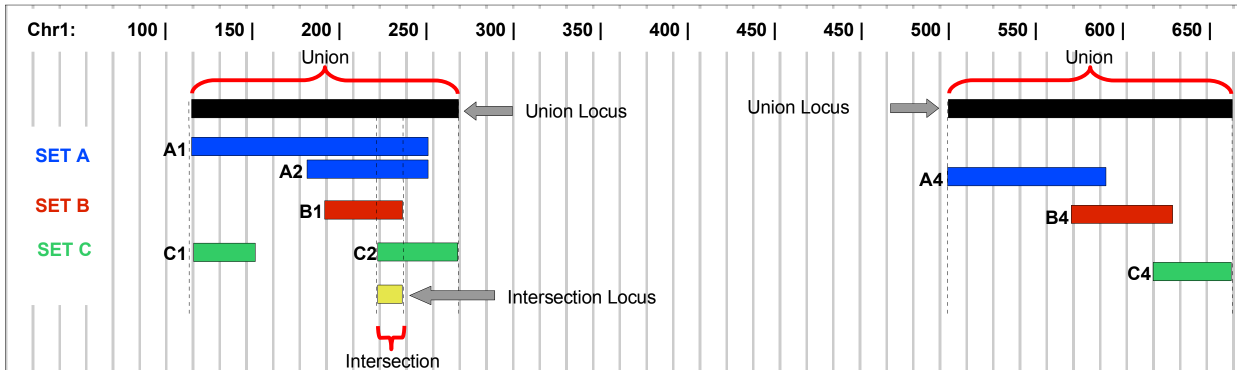
A. Original Loci



B. Result – Overlap of minimum 1 nucleotide, and bridging = *FALSE*



C. Result – Overlap of minimum 1 nucleotide, and bridging = *TRUE*



Results from 2 iterations of LIA.
Fig A represents a set of input data. Figures B and C represent what the results would look like from two slightly different parameter sets – one with bridging and one without bridging.