Table of Contents

Concept

Through the use of cutting edge technology, systems biology has recently begun to emerge as a popular field in the biological community. Experiments in this field result in huge amounts of data, and very often this data represents a group or groups of polynucleotides. These polynucleotides can have many attributes:

- DNA or RNA
- Relative quantities
- Length(s)
- Nucleotide sequence
- Putative function

As a result of the recent Human Genome Project, another attribute is able to be added: **genomic location**. This is a set of coordinates – chromosome, start position, and end position – which represents the point of origin of any nucleotide sequence. This attribute is important because it homogenizes all polynucleotide data and gives us a common attribute across all instances, regardless of source.

Tools have been developed to visualize genomic data, using the coordinates as the common thread. The best example is the Genome Browser at UCSC (http://genome.ucsc.edu/). The UCSC Genome Bioinformatics site acts as the central repository for data related to the Human Genome Project, and provides an outstanding web-based visualization tool for viewing this data.

See Fig 1.

While storage and visualization of genomic data are vital to the progress of the community, *analysis* of this data has not been nearly as well addressed. We believe this is the next frontier of systems biology, and HocusLocus is meant to address this problem. While a very small number of tools exist to perform these kinds of analyses (Penn State's "Galaxy" being the best example - http://g2.bx.psu.edu/), we believe that they've only scratched the surface, and a great deal more functionality and depth is required.

HocusLocus provides a suite of data retrieval and analysis tools which focus on the genomic coordinate attribute of data generated by systems biology experiments. This homogenizing attribute allows us to analyze huge amounts of data from many disparate sources and produce useful and relevant results.

Potentially Novel Items:

1) Utilizing the *genomic coordinate* attribute of systems biology data as a "gold standard". This homogenizes disparate data sets and allows previously impossible questions to be answered via high level analysis.

Data Model

The data analyzed by HocusLocus can come from a variety of sources:

- User loaded data, such as the result of a MicroArray experiment
- Pre-existing data from HocusLocus' own database
- Pre-existing data from 3rd party databases, accessed independently by the user, or via a HocusLocus 'connector'

Data loaded into HocusLocus is converted to a homogeneous data structure, shared by all parts of the system. This data structure is modeled in an object-oriented approach, and is made of up 2 core parts: "Locus" objects and "LocusSet" objects. Each of these is constructed with it's own set of attributes and built-in functionality.

See Fig 2.

Locus

- Attributes:
  - A Locus is the base unit of analyzable data in the HocusLocus system.
  - The only *required* attributes are the genomic coordinates.
  - Remaining core attributes are modeled after the GFF specification (http://www.sanger.ac.uk/Software/formats/GFF/)
  - Any additional attributes can be added dynamically.
  - Locus objects have the ability to be nested in parent/child relationships.
- Functionality:
  - Locus objects define a means by which they can be sorted. Sorting is contextual to their coordinate system (chromosome & position).
  - Locus objects define a means by which they can be compared. Comparisons are contextual to their coordinate system, and result in 'before', 'after', or 'overlap' conditions.
- *See "Locus.html" for JavaDoc specification*

LocusSet

- Attributes:
  - LocusSets are containers for grouping Locus objects. They most often represent an experiment result file, an annotation data set, or any other aggregation of genomic loci.
- Functionality:
  - LocusSets can be dynamically allocated and altered.
  - LocusSets can be merged.
  - LocusSets can affect their contained Locus objects in a global manner, such as sorting or linearizing.
- *See "LocusSet.html" for JavaDoc specification*

Locus sorting is accomplished via Sun's Java specification for object sorting (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html). The Locus object fulfills the specification requirement by implementing the "compareTo" function. A simple conditional logic flow is used, performing a lexicographic comparison of chromosome values and numeric comparison of start position values.

See Fig 3.

Locus comparisons are accomplished by passing a Locus reference to another Locus, and asking "how

do you compare?". The comparison is contextual to the linear coordinate system to which both loci belong (ie: the genome). Possible result values are BEFORE, AFTER, or OVERLAP. The result also depends on what the user considers a *valid overlap* condition. For instance if 2 loci share a common region of only a single nucleotide, do they overlap? Or does the shared region need to be at least 50 nucleotides? The user may instead prefer that a gap of some length is allowed between the 2 loci, and the overlap condition is still true. This flexibility of a true condition is left for the user to define via 2 parameters: "comparison type" and the "comparison value".
The comparison type defines either:

1. FIXED – A specific number of nucleotides defines the overlap/gap criteria. EG: The loci must share a region of at least 50 nucleotides. The loci must be within 1000 nucleotides of each other. Etc.
2. PERCENT – A calculated percentage of the length defines the overlap/gap criteria. EG: The loci must overlap by at least 50%. Etc. (The percentage is calculated using the smaller of the 2 loci).

The comparison value defines:

1. An integer value to accompany the FIXED type.
2. A floating point value to accompany the PERCENT type.

Fig 4 shows a flowchart for the Locus comparison functionality.

LocusSet "flattening" or "linearizing" means that all the Locus objects within a LocusSet are checked for overlap. Loci within the set which overlap (they share a common region of at least 1 nucleotide) are added to a 'parent' Locus. This parent Locus is given a specific type called a REGION, and acts as a container for the overlapping loci. This ensures that all the loci *directly contained* within the LocusSet are linear, but the original data is maintained via the parent/child hierarchy.

See Fig 5.

Potentially Novel Items:

1) Design of the Locus object (and corresponding parent/child hierarchical relationship) to simply and generically represent any and all polynucleotide data.
2) Locus object's "comparison" functionality.
3) LocusSet object's "linearizing" functionality.

Database Schema

As described in the data model, 'Locus' data can come from a variety of sources. Besides the user's own data, one of the main sources of data will be pre-existing databases. HocusLocus will contain it's own database array to serve two purposes:

1. To provide a local, fast lookup of common data sets. This allows the user ease of retrieval, without having to depend on 3rd party sources.
2. To provide specially structured and accessed database tables of additional annotation. This allows the user rapid recovery of the additional data which is normally slow and resource-intensive to generate.

The HocusLocus database array is conceptually modeled after the UCSC database system. The data is structured in a hierarchical fashion, based on species and assembly (version of the genome sequence). For a particular species and assembly, there will be a number of data sets available. Much of the actual data itself is taken directly from UCSC, matching table schema, indexing, and content. Other 3rd party data sources can be leveraged as well. This allows for ease of portability and maintenance, and simply allows for a local copy of this data to be present. However the HocusLocus database array contains a number unique attributes which add to the functionality of the system.

See Fig 6

The HocusLocus database system includes:

- Specialized meta-data tables which describe *what* information is available and *how* it is structured.
- The ability to use these meta-data tables to add new data sets to the system on the fly, and have them become immediately available.
- Uniquely structured tables of additional annotation, allowing for rapid retrieval of large repositories of information with minimal overhead.

The meta-data layer consists of a main database which acts as a central point of access, and contains information describing the remainder of the array. Each species and assembly combination is housed in it's own database, and specific tables in the “main” database list what combinations are available. Other meta-data describes how to access those data sets, as well as global table structure descriptions for each unique set of content taken from UCSC.

As mentioned previously, each species/assembly database contains some number of data sets gathered from 3rd party sources such as UCSC or others. When describing this data, the **genomic location** attribute (chromosomal coordinates) is the focus of the HocusLocus system. However there are many other attributes of importance – the most common being the sequence – that may be required as part of the analysis. The HocusLocus database array provides a means by which some of this information can quickly and easily accompany the loci in a data set. Currently the two additional annotation sets provided are nucleotide sequence, and phylogenetic conservation. In each case, an attribute of each nucleotide must be maintained: a sequence 'letter' (A,T,C,G,etc), or a conservation score. Each table is structured in a similar manner: The attributes of each nucleotide are grouped together into equal length short segments, and each segment is given it's corresponding chromosomal position. In this case only the the chromosome and first nucleotide (start position) are tracked. An index is also created based on the chromosomal coordinates, thus giving a unique index. In this way, data that was previously very “horizontal” - for instance an entire chromosome sequence – is transformed into easily indexable “vertical” data. This allows us to draw upon the power of the database engine to perform extremely fast retrieval of large amounts of information.

Certainly many other retrieval-speed problems still exist, such as disk access speeds, data caching, network traffic, etc. However the HocusLocus storage schema allows us to all but eliminate the bottleneck of *seek time*, while allowing all the benefits of storing your raw data in a relational database.

See Fig 7

Potentially Novel Items:

1) Database table structure and accompanying functionality for rapid retrieval of annotation data (as described above and in Figure 4).

System Overview & Workflow

Design:

HocusLocus is designed as a traditional 3-tier system utilizing a relational database, a web based application server, and web browser clients.

See Fig 8

The database array is implemented using MySQL version 5. The databases, each contextual to a species and assembly (as described previously), all reside within a single instance of the database engine. This instance can reside at any location that is network accessible from the application server. A JDBC connection is used to link the application server to the database. A special sub-system module called the "HLDBM" (HocusLocus Database Manager) is responsible to for all database access from the system. This provides a single point of access and control over all database processes.

The application server is implemented using standard J2EE technologies (Servlets and JSPs) on Jakarta Tomcat version 5. User interaction is session based, however it will be possible to store a session state at the server for later retrieval. A "Model-View-Controller" design pattern is used to control interaction and data flow within the system. The model is the current set of data and state information for a user session. It is made up of LocusSet objects (as described previously) representing user-loaded and pre-existing database data, as well as new data sets generated during the session. The model also holds all session state information such as algorithm parameters and process cardinality. The controllers are the individual system tools which act as independent modules within the system. Each module represents an algorithm implementation – intersection analysis, control generation, etc. - that can be executed individually or in succession. The view is a portion of the client web page which gives a visual representation of the data sets and session state. A flow diagram is used to show:

- Each data set and it's annotation.
- Each instance of a module used to process the data, along with the parameters used.
- Relationships among the data sets and processes describing the interactions.

See Fig 9

The HocusLocus client is based in JSP technology and can be used from any compatible web browser. It presents the user with a menu of operations they can perform, such as uploading data, retrieving additional data from a database, or executing an analysis process on the data. There is also a section of the interface for user input that may be required for a given operation. This area is context sensitive, and presents appropriate options for currently selected operation. Finally, the content area of the interface presents the user with a view of their data and operations performed, as described above. This information is rendered as a flowchart, sequentially describing each data set and the operations that were performed on them. The user can interact with this diagram to easily obtain more detailed information about any of the elements, download data sets, or to generate an image file for documentation purposes.

Usage:

In order to utilize the capabilities of HocusLocus, a user's data must first contain the genomic coordinate attribute. As described in section 1 "Concept", this attribute often exists by default as part of the result of an experiment. However this feature may not be implicit for some technologies (for example: some MicroArray results may provide accession numbers only, or require significant statistical analysis before coordinates can be generated). In these cases, HocusLocus can provide means to transform the data:

- Modules can be provided to perform data lookup – such as accession number to Locus.
- Novel algorithms/modules can be provided to perform statistical analysis.
- 3rd party algorithms can be integrated into the system.
- HocusLocus can “link out” to a 3rd party web service for data conversion.

Once a user's data contains genomic coordinates, it is then loaded into the system. Additional data sets can added from the HocusLocus database if desired. The user then chooses which operations they want to perform on which data sets, and resultant data sets are generated. Since all the data sets are homogeneous, they can be they can be mixed and matched in any operation and in any order. The sequence of operations, data sets generated, parameters used, and all other corresponding information is displayed in the client flow diagram. The user can continue to perform analyses until the desired resultant data set is generated.

A sample flowchart of a complete user interaction is presented in *Figure 10*.

Potentially Novel Items:

1) The “flow diagram” representation of all the user's current data sets, the processes performed on those data sets, and all accompanying parameters and information.

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 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 the "Data Structures" section).

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 the "Data Structures" section (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:

Each LocusSet given to LIA is prepped before the algorithm instantiates. First the LocusSets are copied, in order to preserve the integrity of the original sets. Then the they are ordered, as described in the "Data Structures" section. Lastly the LocusSets are linearized, 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 linearization 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 11.

For example in Figure 11, 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 12A, when comparing A1, B1, and C1, B1 and C1 do not share a common region and the condition is considered negative without bridging - Fig 12B. However if bridging is allowed, locus A1 bridges B1 and C1, and the condition is considered positive - Fig 12C. 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 12C:

Set A	***Set B***	***Set C***
A1	B1	C1
A1	B1	C2
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 12C.

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.

Usage:

Forthcoming...

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 $(X \in A) \cap (Y \in B) \cap \ldots$
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.