Pantzerfaust

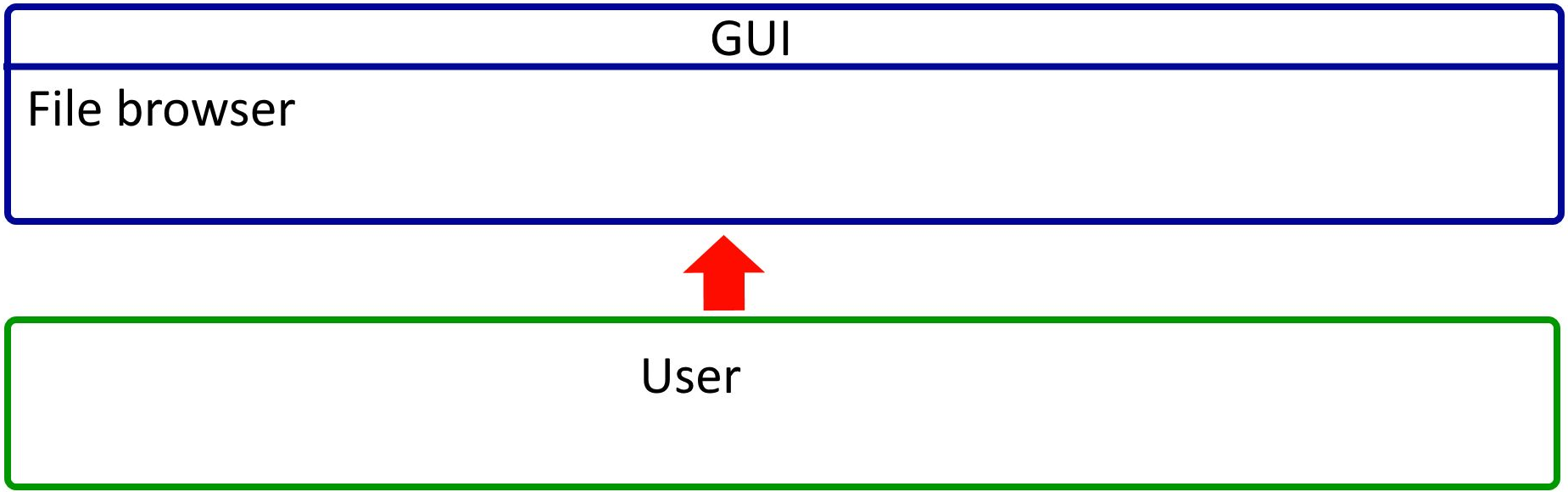
ProgrammingLife 2016 - Technical Customer Reflection

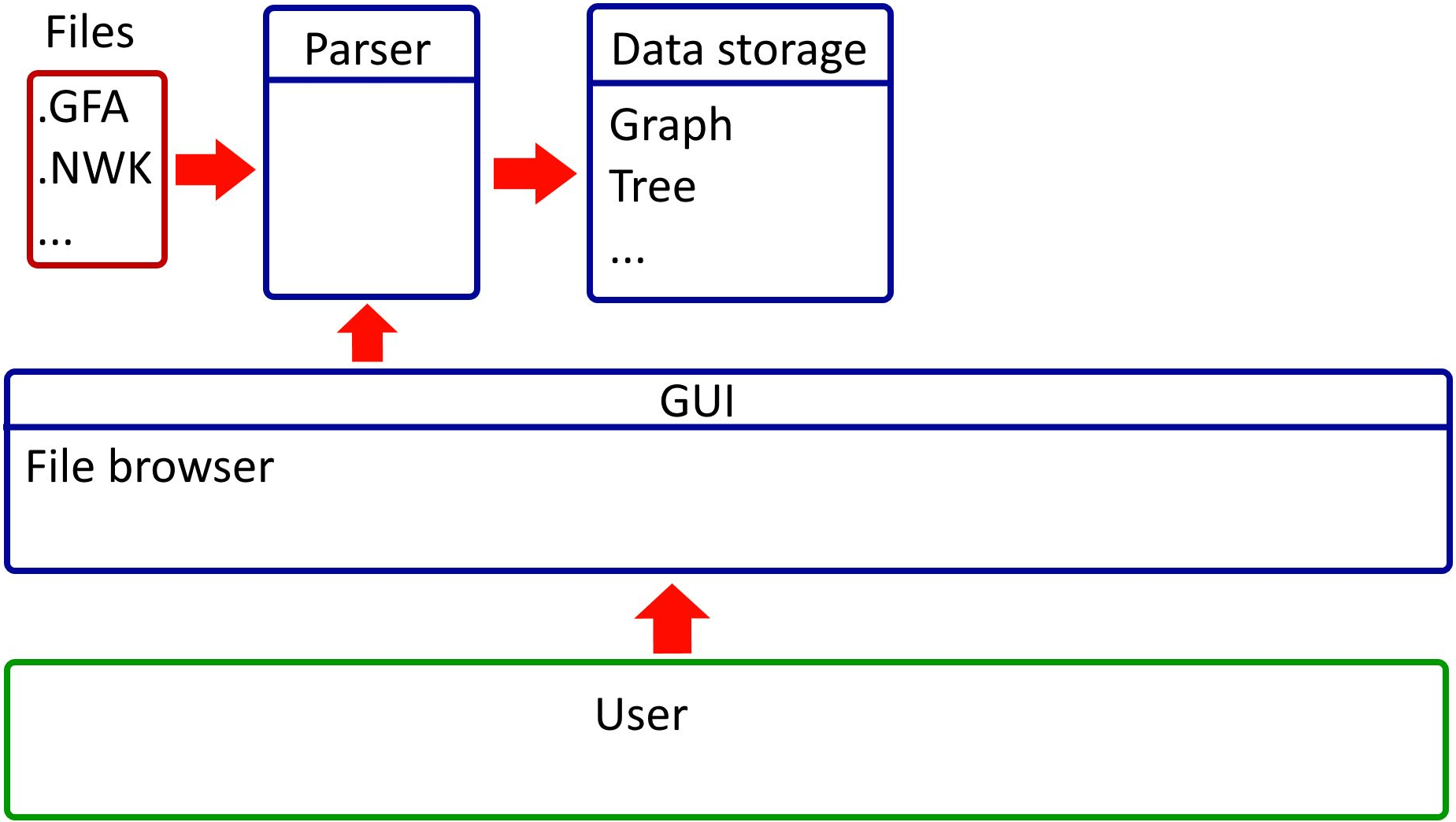
Overview

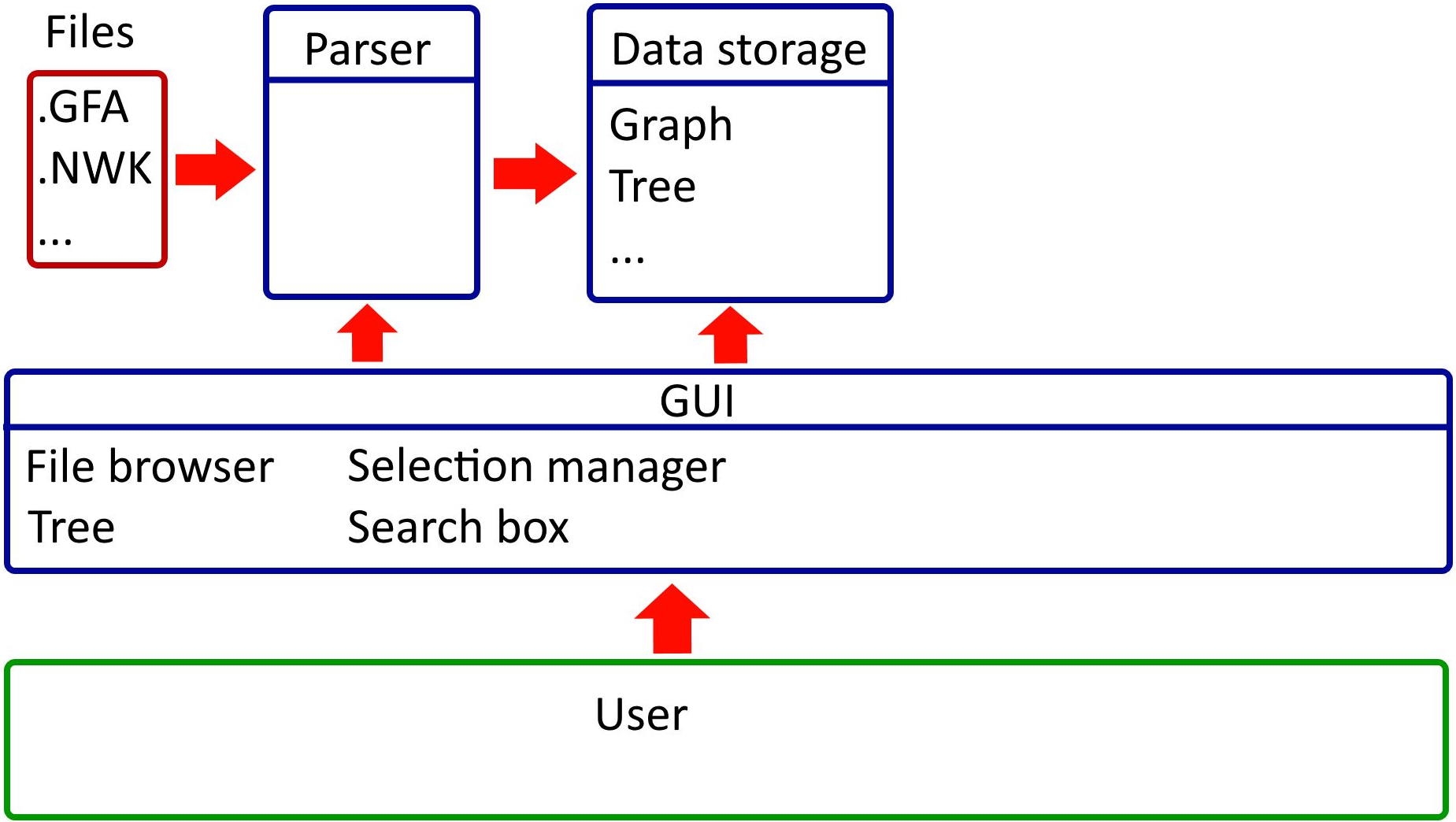
- Application architecture
- Internal data representation
- Visual components and scalability
- Semantic zoom strategy
- Technology stack
- General recommendations

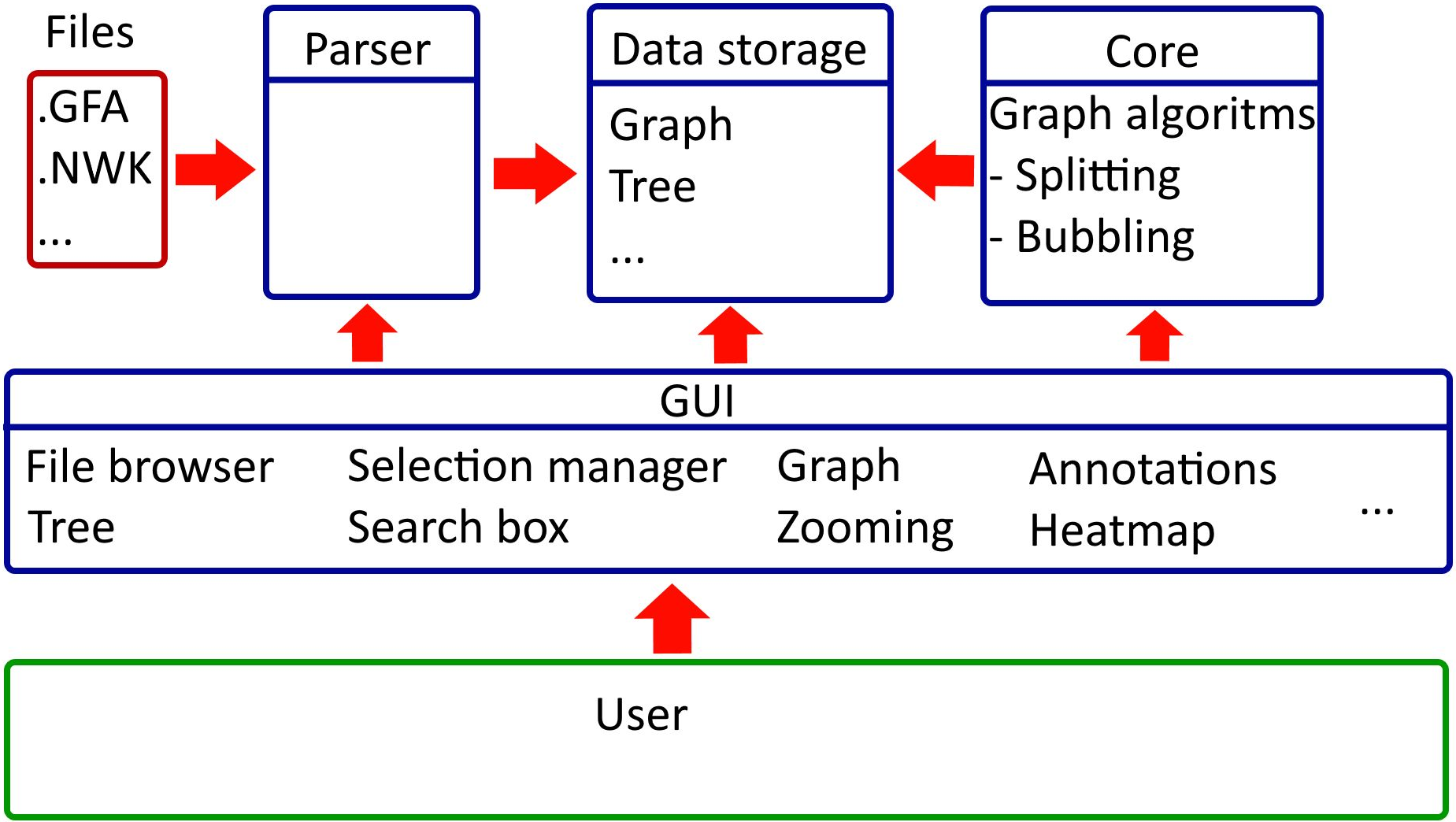
Application architecture

- GUI
- Parser
- Data storage
- Core


GUI
File browser
User


Files
.GFA
.NWK
...
Parser
Data storage
Graph
Tree
...
GUI
File browser
User


Files
.GFA
.NWK
...
Parser
Data storage
Graph
Tree
...
GUI
File browser
Tree
Selection manager
Search box
User


Files
.GFA
.NWK
...
Parser
Data storage
Graph
Tree
...
Core
Graph algoritms
- Splitting
- Bubbling
GUI
File browser
Tree
Selection manager
Search box
Graph
Zooming
Annotations
Heatmap
...
User

Application architecture

Good

- Simplicity
- Loosely coupled modules

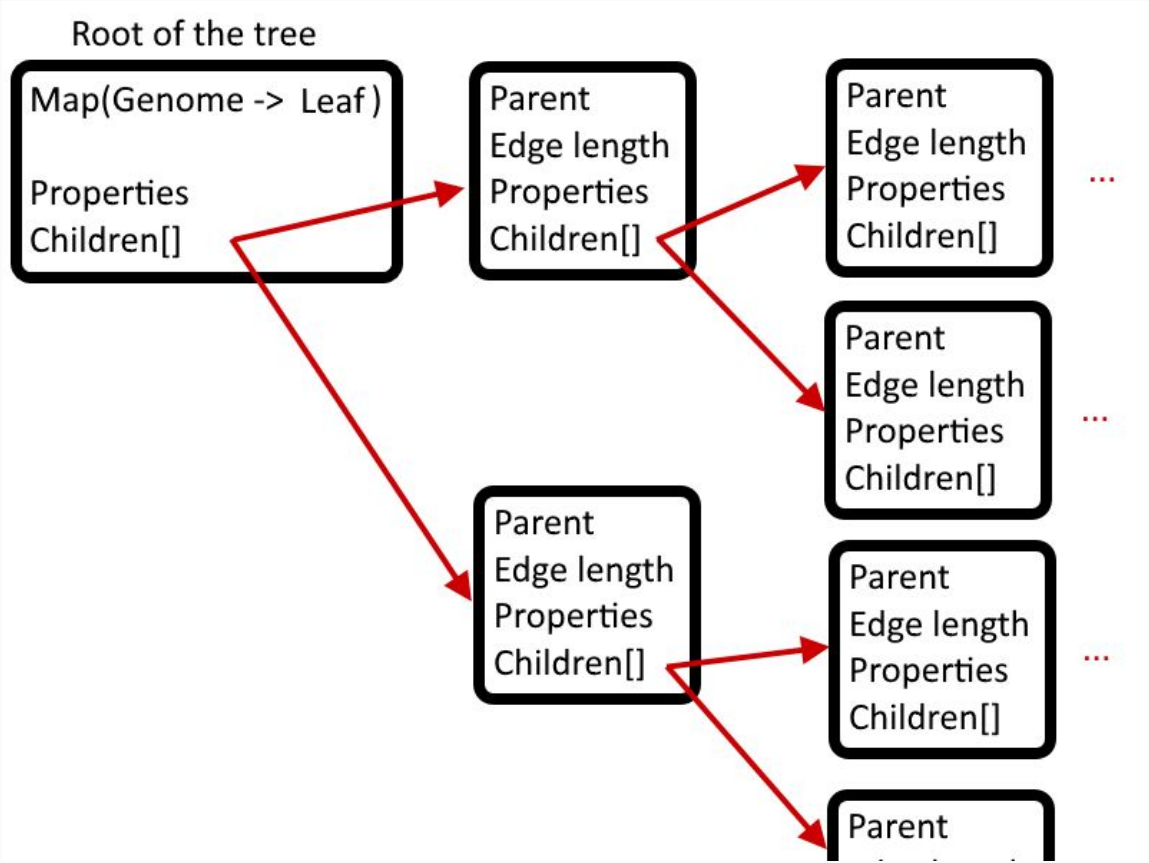
Could be improved

- Some of our GUI files have a high complexity. It would’ve been better to make them smaller

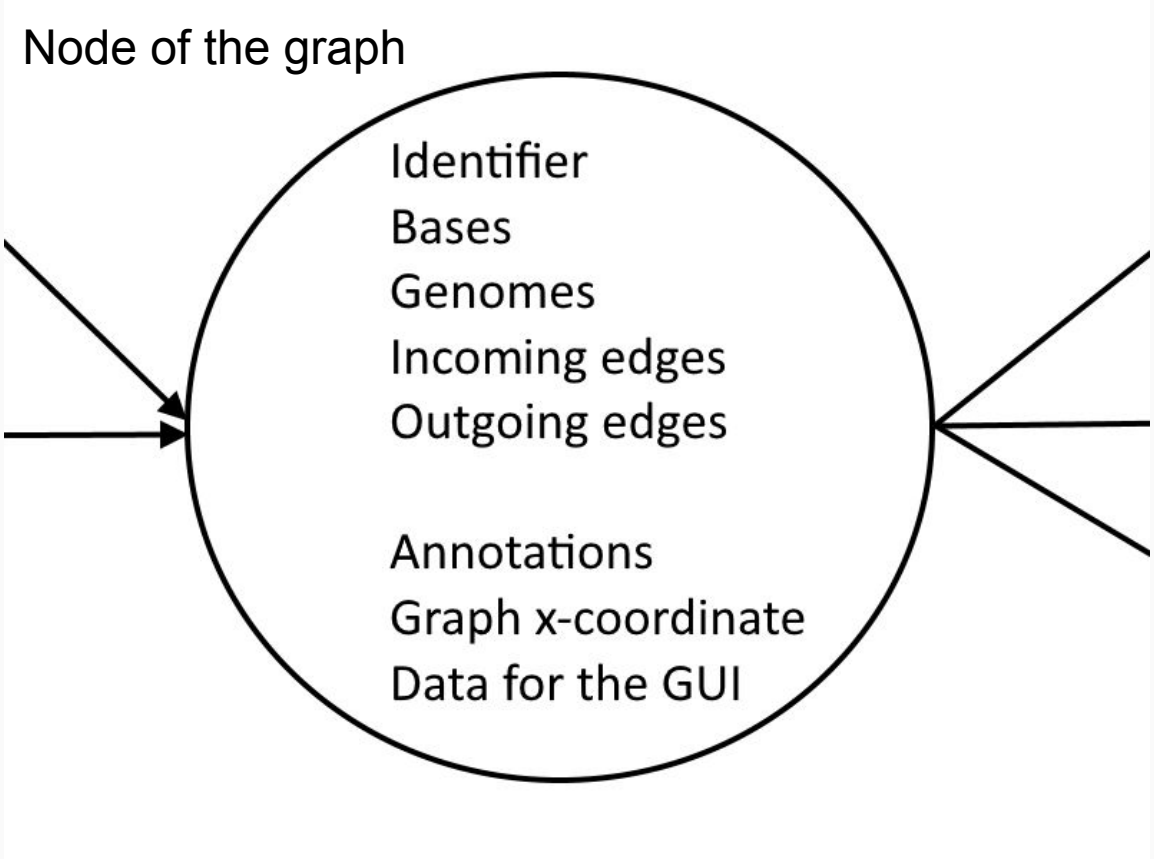
Internal data representation

- Metadata
- Annotations
- Phylogenetic tree
- Genome graph

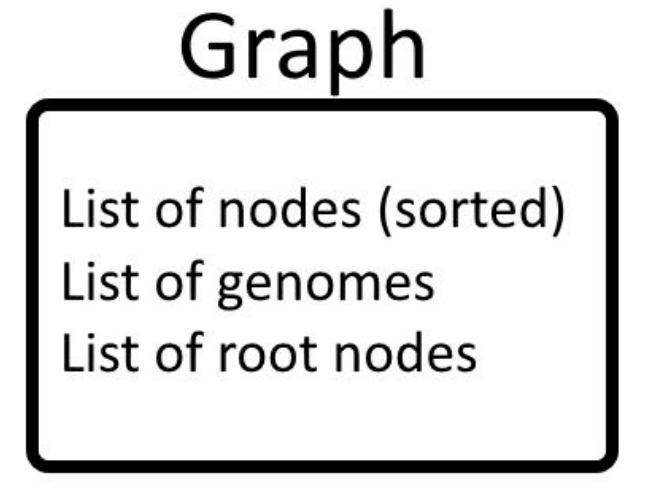
Phylogenetic tree

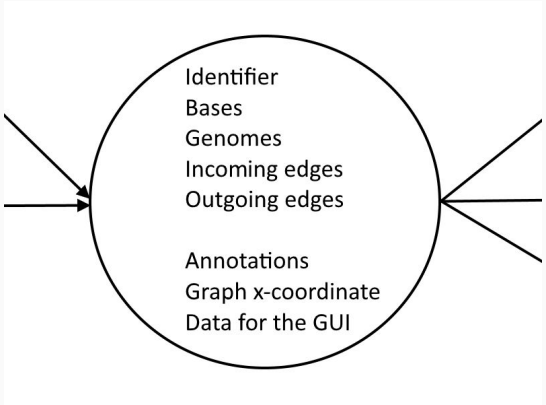


Graph - Node Object



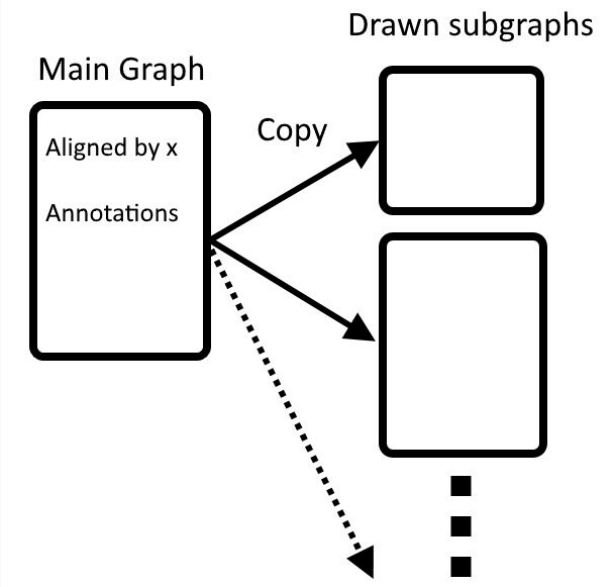
Graph - Graph Object

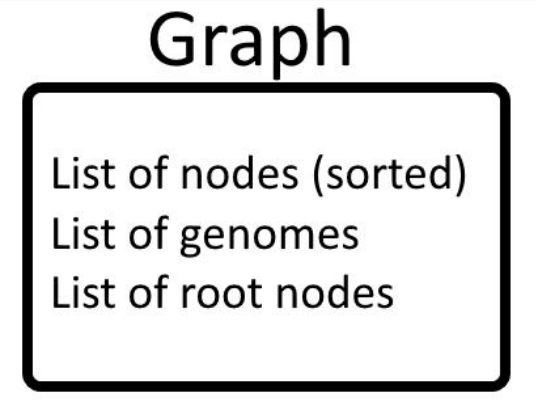


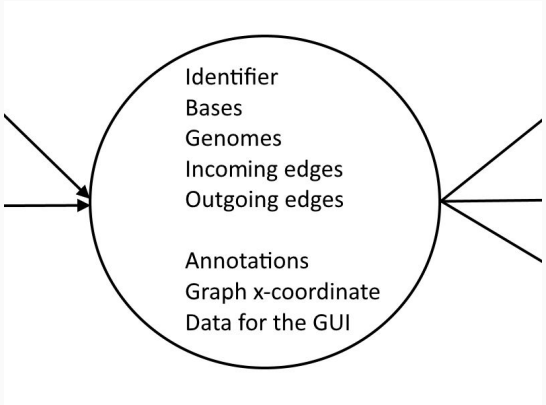


Graph - Main Graph

- Sorted by x-coordinate
- Annotations







Graph - Optimizations

- Memory
  - Genomes - id instead of string
  - Bases - 3 bits encoding*
- Performance
  - Use hashmaps/hashsets
  - Sort lists which will be compared often
- Main tradeoff:
  HashMap (36 bytes/value***) VS ArrayList (4 bytes/value**)

* https://github.com/ProgrammingLife2016/PL2-2016/blob/dev/PL2/PL2-shared/src/main/java/nl/tudelft/pl2016gr2/model/graph/data/BaseSequence.java
** http://java-performance.info/memory-consumption-of-java-data-types-1/
*** http://java-performance.info/memory-consumption-of-java-data-types-2/

Internal data representation

Good

- Don’t create objects for each edge - storing edges is very expensive
- Do most computations on the main graph and copy the computed values when drawing a subgraph

- Store genomes as efficiently as possible for each node
- Store bases as efficiently as possible
- Use a profiler to check the memory consumption
- Use HashSets/HashMaps if the memory increase isn’t too high
- Sort lists which will often be checked for equality. Comparing two sorted lists takes O(n) time while comparing unsorted lists takes O(n2) time.

Could be improved

- Genomes could have been stored for each edge in a node instead of for the node itself. This doesn’t cost much more memory and calculating which genomes travel over which edges is expensive.
- Data could be written to a database, so less RAM is needed. This is however pretty complex to implement efficiently and should only be needed for very large datasets. Simply using a powerfull desktop with 128GBs of RAM should be enough to load around 40K-100K genomes (performance would have to be improved though).

Visual components and scalability

- Our performance (328 graph):
  - Nodes drawn: around 50 - 600
  - Edges drawn: around 2,000 - 16,000
  - FPS: 20 - 60 while zooming*
  - CPU: below 20% while zooming*
  - Memory: around 500MB

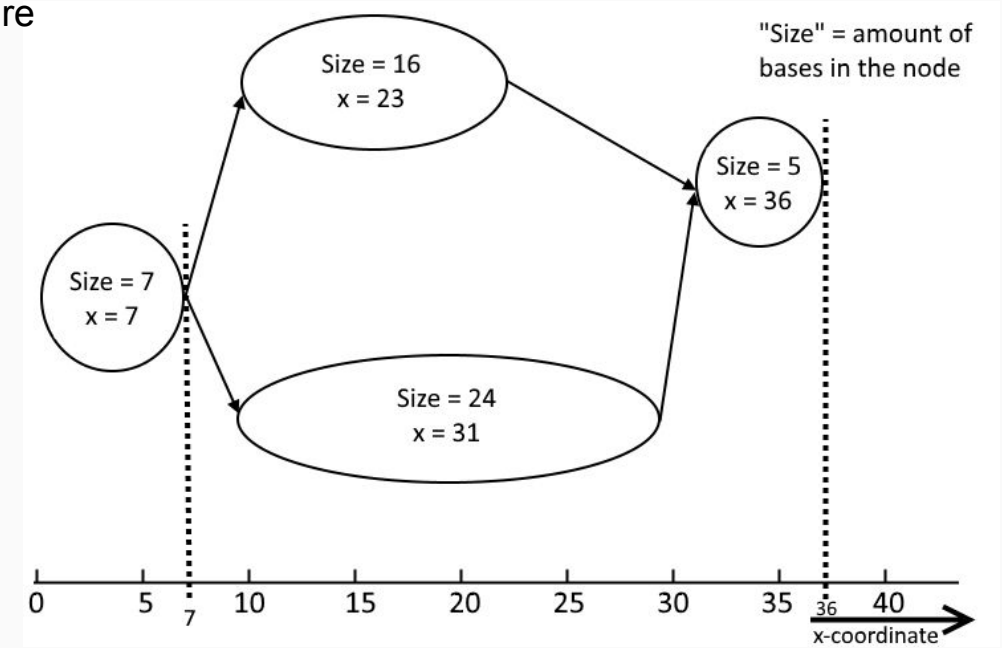
* With an i7-5820K (6-Core @3.3GHz) and GeForce GTX 960
See the resources for a profiling session

Drawing - Scalability

- Only draw nodes which you can see
- Redraw instead of moving
- Precompute complex computations
- Edges -> canvas

Drawing - horizontal position

- Linear scale
- Custom coordinate system, see picture
- Easy to precompute



Visual components and scalability

Good

- Draw non-interactive components in a canvas (edges)
- Only create JavaFX object for elements which can be seen
- Don’t use complex visual elements (such as gaussian blurs)
- A linear coordinate system makes things easier (in our case)

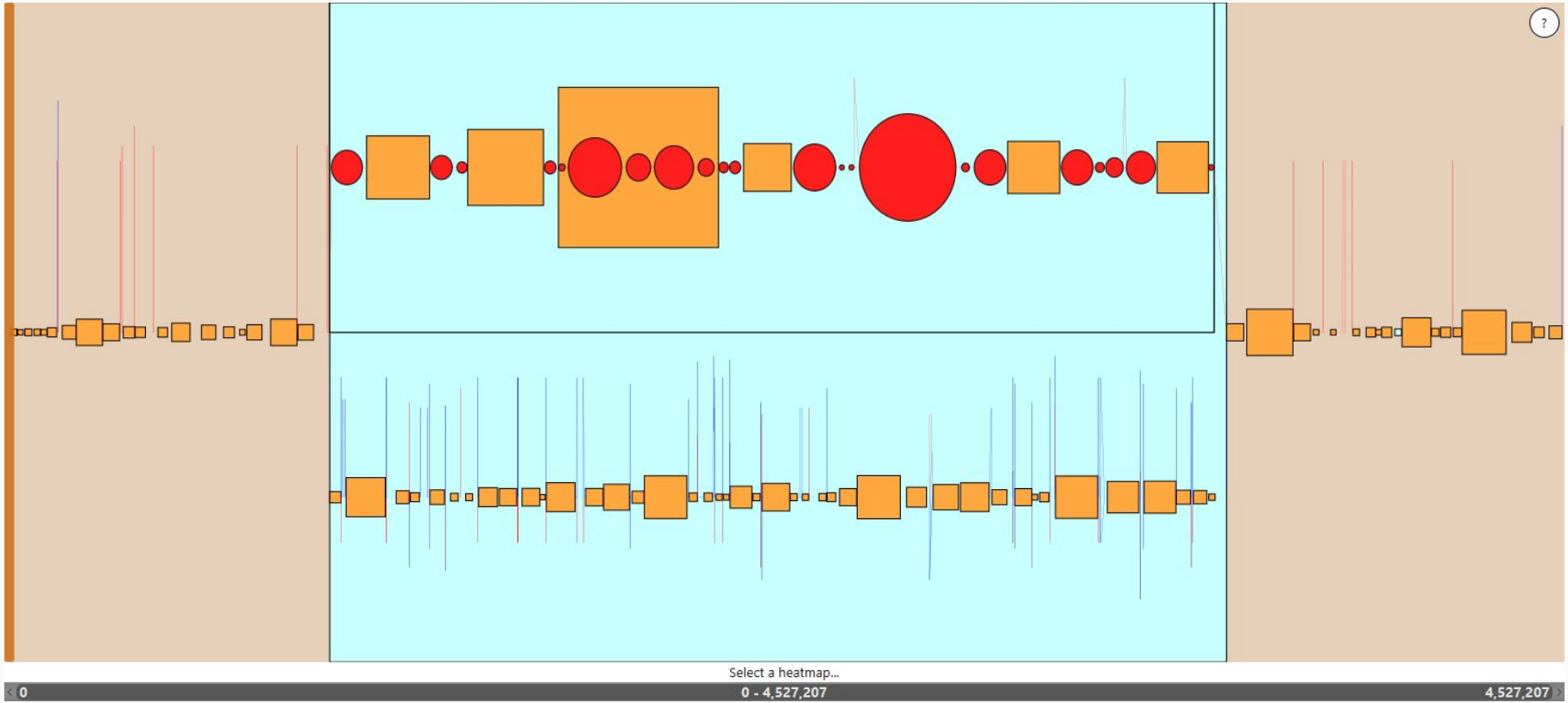
Could be improved

- Better memory management of JavaFX objects: remembering and reusing old JavaFX objects will make the life of the garbage collector a lot easier.

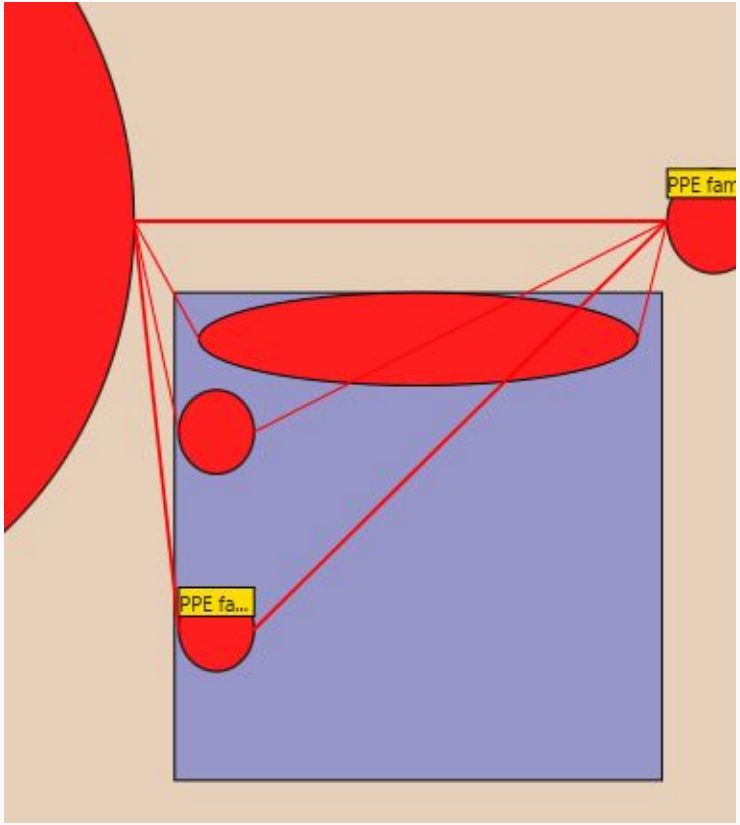
Semantic zoom strategy

- Semantic -> Bubbles
  - Phylogenetic
  - Graph (source - sink pairs)
  - Point mutation
  - Indel
- Non-Semantic -> change x-coordinates

Bubble - Phylogenetic



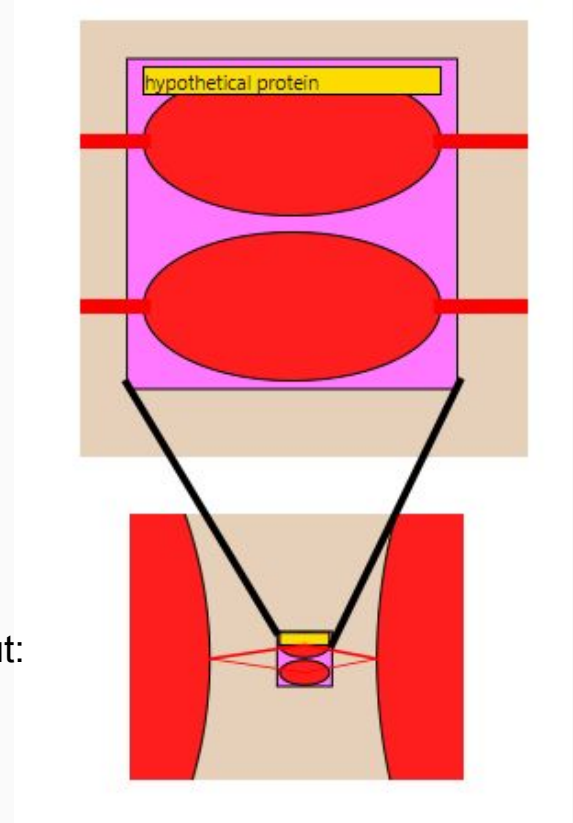
Bubble - Graph (source-sink pairs)



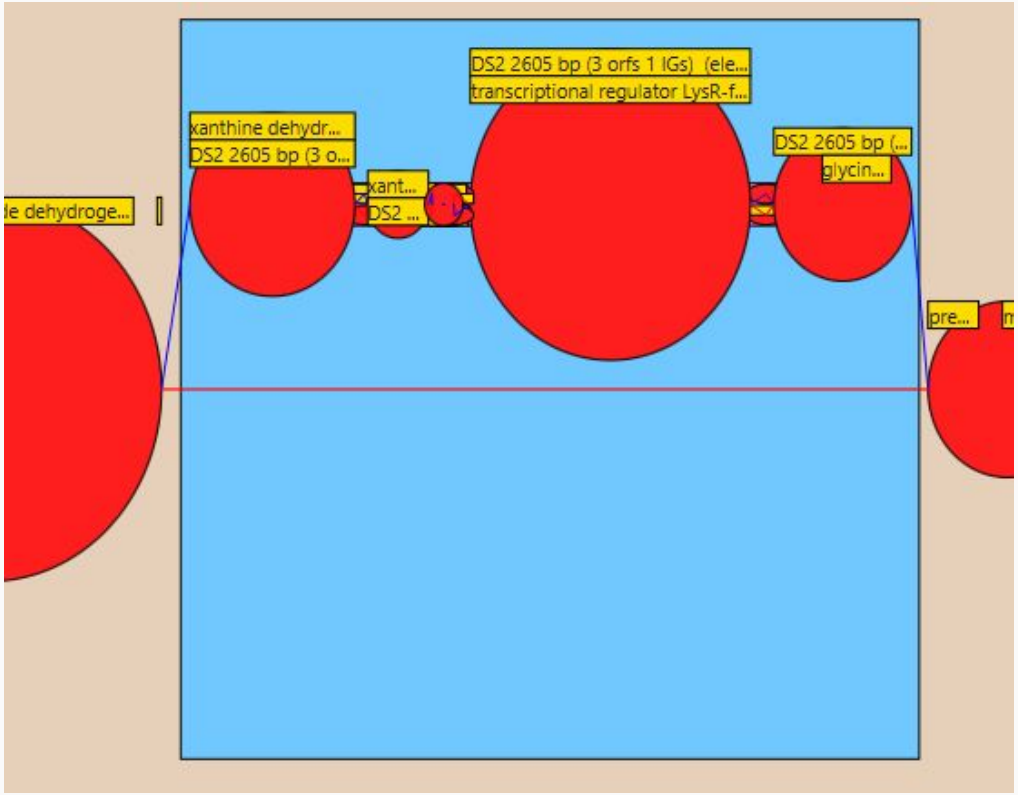
Bubble - Point mutation

Zoomed in:

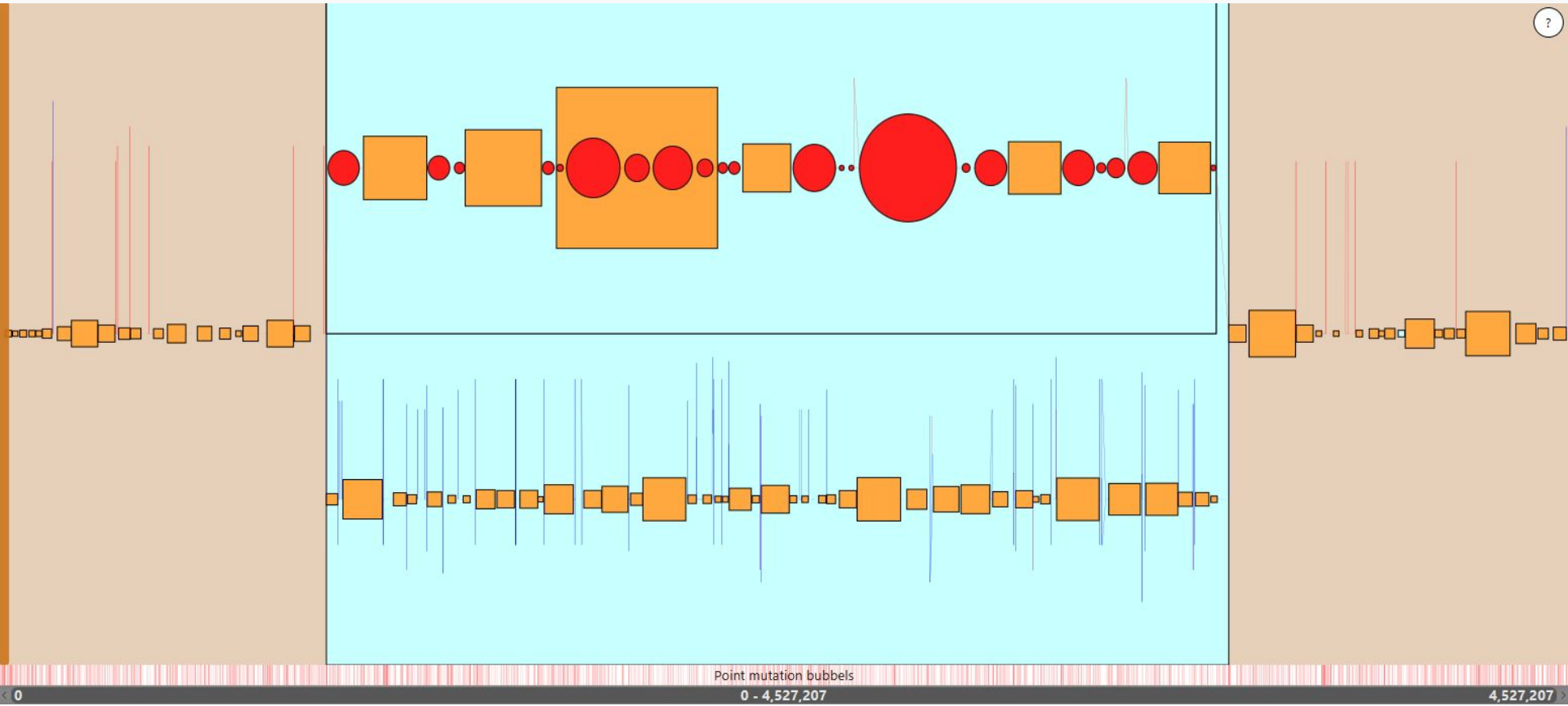
Zoomed out:



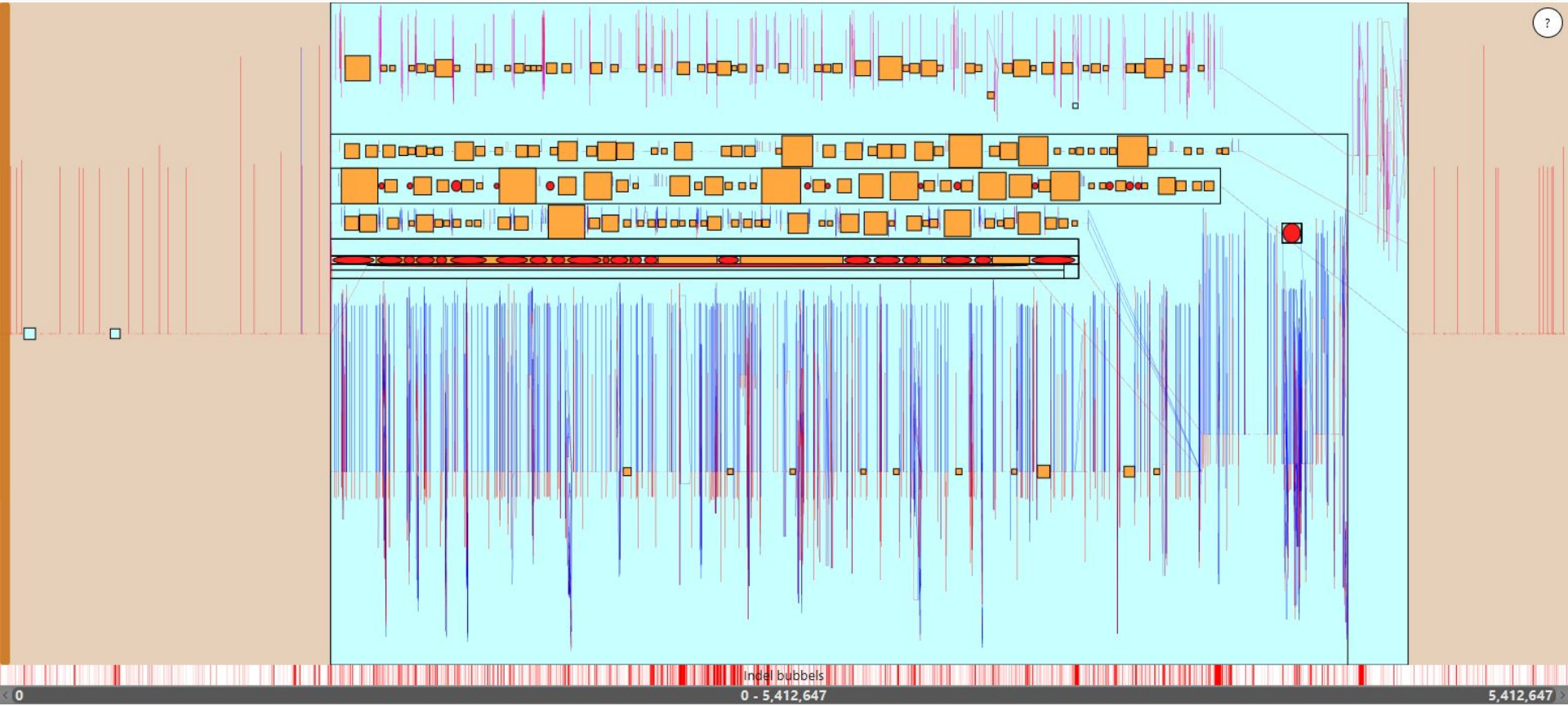
Bubble - Indel



Point mutation heatmap

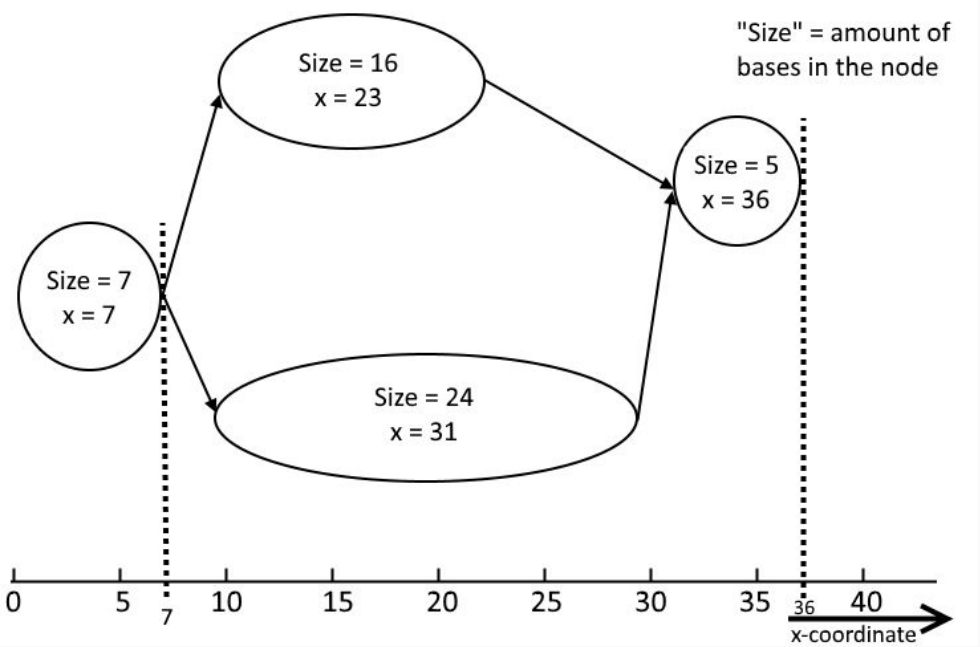


328 graph with indel heatmap



Non-Semantic Zoom

- Change range of drawn x coordinates



Semantic zoom strategy

Good

- Smooth zooming
- Smooth semantic zooming
- Easy to identify mutations
- Nice graph heatmap integration

Could be improved

- Very small nodes (with fewer than 150 bases) are drawn larger, thus they overlap.
- When zoomed in a logarithmic scale might be more useful. This could be done by implementing a way to toggle between linear and logarithmic scaling.

Technology stack

- Java
- JavaFX

Technology stack

Good

- Everyone was familiar with the used technologies
- Very few different components: easy to test/debug
- No communication overhead between different technologies

Could be improved

- A database could be used to store parts of the memory which aren't required at the moment. This would reduce the used RAM.
- Some computations could be executed in lower level languages, or even on the GPU, to improve performance.

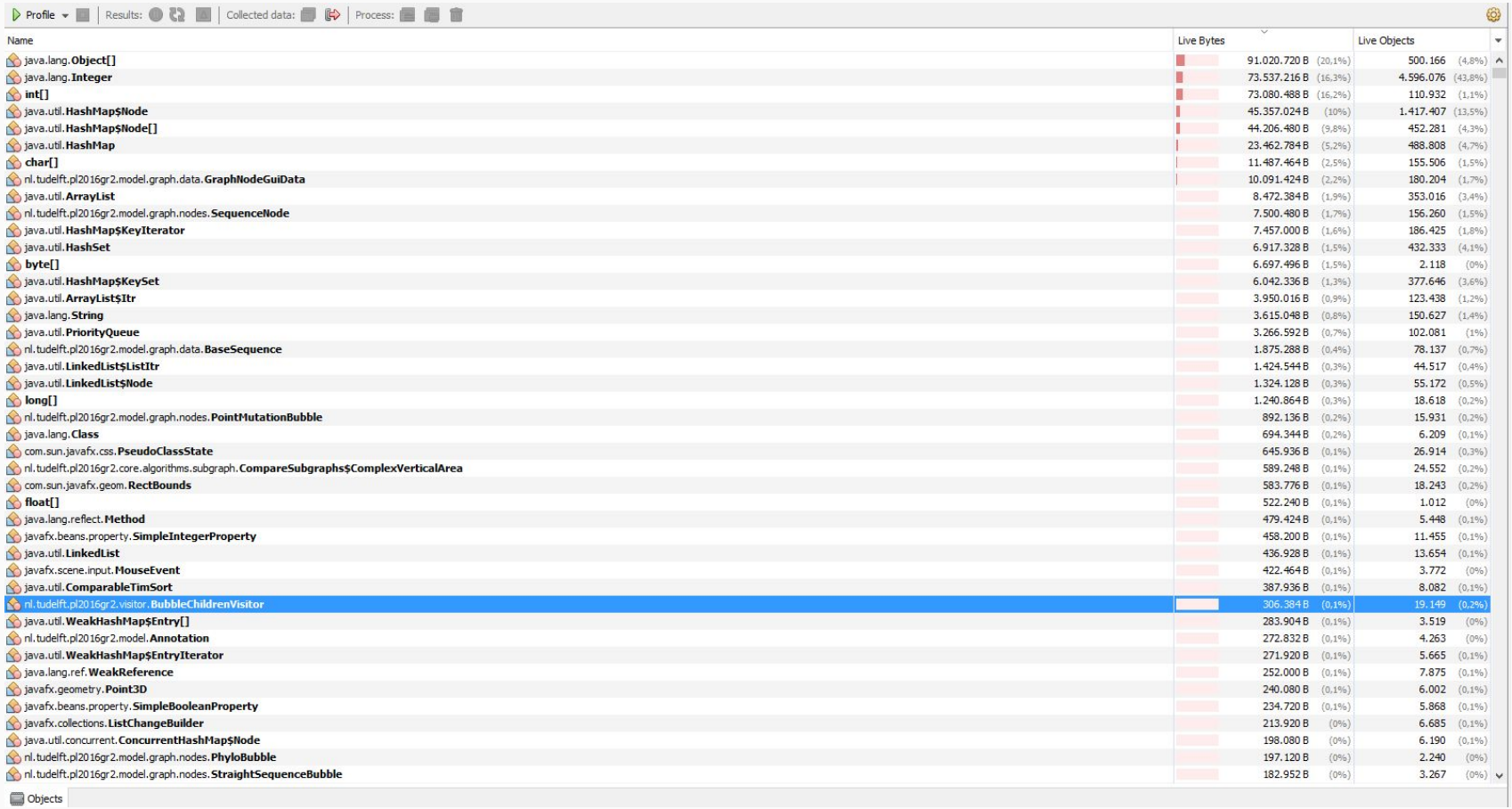
General recommendations

- First focus on the horizontal alignment (topological)
- Don’t start too late with the vertical alignment of the graph
- Smooth zooming + popping requires a linear x-scale (?)
- Start with the phylogenetic tree
  - Select based on tree
  - Use tree to search
- Learn how to use a profiler to profile CPU and memory usage (see resources)

Resources

Profiler - CPU usage when loading and drawing the 328 graph (netbeans profiler)

Profiler - memory usage with 328 graph loaded & drawn (default netbeans profiler)



Profile | Results: | Collected data: | Process:

Name	Live Bytes		Live Objects	
java.lang.**Object[]**	91.020.720 B	(20,1%)	500.166	(4,8%)
java.lang.**Integer**	73.537.216 B	(16,3%)	4.596.076	(43,8%)
int[]	73.080.488 B	(16,2%)	110.932	(1,1%)
java.util.**HashMap$Node**	45.357.024 B	(10%)	1.417.407	(13,5%)
java.util.**HashMap$Node[]**	44.206.480 B	(9,8%)	452.281	(4,3%)
java.util.**HashMap**	23.462.784 B	(5,2%)	488.808	(4,7%)
char[]	11.487.464 B	(2,5%)	155.506	(1,5%)
nl.tudelft.pl2016gr2.model.graph.data.**GraphNodeGuiData**	10.091.424 B	(2,2%)	180.204	(1,7%)
java.util.**ArrayList**	8.472.384 B	(1,9%)	353.016	(3,4%)
nl.tudelft.pl2016gr2.model.graph.nodes.**SequenceNode**	7.500.480 B	(1,7%)	156.260	(1,5%)
java.util.**HashMap$KeyIterator**	7.457.000 B	(1,6%)	186.425	(1,8%)
java.util.**HashSet**	6.917.328 B	(1,5%)	432.333	(4,1%)
byte[]	6.697.496 B	(1,5%)	2.118	(0%)
java.util.**HashMap$KeySet**	6.042.336 B	(1,3%)	377.646	(3,6%)
java.util.**ArrayList$Itr**	3.950.016 B	(0,9%)	123.438	(1,2%)
java.lang.**String**	3.615.048 B	(0,8%)	150.627	(1,4%)
java.util.**PriorityQueue**	3.266.592 B	(0,7%)	102.081	(1%)
nl.tudelft.pl2016gr2.model.graph.data.**BaseSequence**	1.875.288 B	(0,4%)	78.137	(0,7%)
java.util.**LinkedList$ListItr**	1.424.544 B	(0,3%)	44.517	(0,4%)
java.util.**LinkedList$Node**	1.324.128 B	(0,3%)	55.172	(0,5%)
long[]	1.240.864 B	(0,3%)	18.618	(0,2%)
nl.tudelft.pl2016gr2.model.graph.nodes.**PointMutationBubble**	892.136 B	(0,2%)	15.931	(0,2%)
java.lang.**Class**	694.344 B	(0,2%)	6.209	(0,1%)
com.sun.javafx.css.**PseudoClassState**	645.936 B	(0,1%)	26.914	(0,3%)
nl.tudelft.pl2016gr2.core.algorithms.subgraph.**CompareSubgraphs$ComplexVerticalArea**	589.248 B	(0,1%)	24.552	(0,2%)
com.sun.javafx.geom.**RectBounds**	583.776 B	(0,1%)	18.243	(0,2%)
float[]	522.240 B	(0,1%)	1.012	(0%)
java.lang.reflect.**Method**	479.424 B	(0,1%)	5.448	(0,1%)
javafx.beans.property.**SimpleIntegerProperty**	458.200 B	(0,1%)	11.455	(0,1%)
java.util.**LinkedList**	436.928 B	(0,1%)	13.654	(0,1%)
javafx.scene.input.**MouseEvent**	422.464 B	(0,1%)	3.772	(0%)
java.util.**ComparableTimSort**	387.936 B	(0,1%)	8.082	(0,1%)
nl.tudelft.pl2016gr2.visitor.**BubbleChildrenVisitor**	306.384 B	(0,1%)	19.149	(0,2%)
java.util.**WeakHashMap$Entry[]**	283.904 B	(0,1%)	3.519	(0%)
nl.tudelft.pl2016gr2.model.**Annotation**	272.832 B	(0,1%)	4.263	(0%)
java.util.**WeakHashMap$EntryIterator**	271.920 B	(0,1%)	5.665	(0,1%)
java.lang.ref.**WeakReference**	252.000 B	(0,1%)	7.875	(0,1%)
javafx.geometry.**Point3D**	240.080 B	(0,1%)	6.002	(0,1%)
javafx.beans.property.**SimpleBooleanProperty**	234.720 B	(0,1%)	5.868	(0,1%)
javafx.collections.**ListChangeBuilder**	213.920 B	(0%)	6.685	(0,1%)
java.util.concurrent.**ConcurrentHashMap$Node**	198.080 B	(0%)	6.190	(0,1%)
nl.tudelft.pl2016gr2.model.graph.nodes.**PhyloBubble**	197.120 B	(0%)	2.240	(0%)
nl.tudelft.pl2016gr2.model.graph.nodes.**StraightSequenceBubble**	182.952 B	(0%)	3.267	(0%)

Objects

Profiler - time spent in each method (default netbeans profiler)