# Desirable Future Features for Galant (annotated).

**Red and bold are high priority (must be done or close to being done). *Bold and italic are secondary, but still important (there must be an indication in the developer documentation as to which part of the code should be modified and, if possible, a workaround).*** ***Green*** ***and bold-italic indicate high priority for next release, but require significant design changes.*** Annotations give location(s) of source code that might need to be modified to implement the feature in question. All source code is in subdirectory src/edu/ncsu/csc/Galant/ of the git repository spring2013Team08/. The prefix is omitted, as are prefixes of subdirectories containing the sources. The .java extension is also omitted. All annotations are in the form of comments.

* **Programmer Documentation** – Start with an outline of an algorithm with an example. The documentation should include all methods and macros in the API as well as an indication of the status of their implementation; ones that are desirable but have not been implemented yet or only partially implemented with a workaround should be listed if they were discussed.
* **Developer Documentation** – should include Javadoc and text description with UML. The items in the programmer documentation and user documentation should be addressed (where located in the code)
* **User Documentation** – should explain how to draw a graph, load and save files, set preferences, etc.

### Editor

* An undo capability in the editor, at least for last operation.
* Other standard editor functions such as search/replace. Since editing can be done offline these may not be important. The Galant editor can be used primarily to avoid an edit-save-open-compile cycle.
* Would be nice to highlight selected nodes and edges in the graphml text.
* An editor command to change the direction of a directed edge.

### Input/Output

* Allow file names on the command line, as in

java –jar Galant.jar –graph <graph> -alg <algorithm>

* ***Make reading of GraphML attributes contingent on a graph type declared at the beginning of the file,*** e.g., set layer and position\_in\_layer attributes for nodes only if the graph is layered and call on the LayeredGraph constructor in that case. This also allows for LayeredGraph (and future specialized graph types) to extend the Graph class.
* Make all changes during algorithm execution, including the addition/deletion of vertices and edges ephemeral and (a) allow saving of snapshots; and (b) prompt user whether or not to save when exiting algorithm (it’s not clear what should be saved, however).
* Make it possible to export graphs (and snapshots) to, e.g., pdf or, better yet a drawing program. The right strategy here is probably a standalone program that converts GraphML to other formats.
* Persist opened files through edit sessions and don’t allow opening of the same file in two different windows (actually emacs does allow this but I only use it for large source files).
* Query before closing if file has not been saved.

### Graphs (logical attributes)

* It would be good to have Node methods *getFirstOutgoingEdge, getFirstIncidentEdge* and *getFirstIncomingEdge* for sorting algorithms, where the nodes are presumably on a single directed path.
* Use node id’s as provided by the input GraphML file. This requires
  + a semantics when two nodes with the same id are encountered (use last one?)
  + a method *maxNodeId()* that may return a value distinct from the number of nodes
* When a graph is parsed, it would be nice to have attributes other than those defined by Galant stored for the nodes and edges. There is already a mechanism for creating arbitrary attributes: why not use it?

### Compilation and execution

* Friendlier error reporting at both compile and runtime:a separate window for error messages (that pops up at the appropriate time), avoid full stack trace, identify compile-time errors in terms of algorithm rather than Java syntax. Highlighting of offending line for errors would be ideal.
* ***A more sophisticated preprocessor for the compiler*** that would, aside from expanding macros, also protect the programmer from Java-specific errors and give error messages that make sense in the context of the original program.

For example:

* + put automatic semicolons at the ends of lines where appropriate
  + **allow global variables and global creation of data structures without use of key words such as static and final**
  + provide alternate names (for programmers who either forget or want the algorithm to be commensurate with its context) and non-OO invocations of methods
  + automatic conversion of nodes and edges to their integer id’s (to use as array indices) – can there be a toInteger() method?
* There are a number of possible execution options, come of which don’t allow for backtracking; see the **Algorithm** section below for a more comprehensive discussion.
* There’s a problem with heap space if there are too many changes in state. Maybe this can be streamlined so that state changes are tied to beginStep() and endStep() calls.

### Display attributes: selection, weights and labels

* Labels and weights on edges should be hidden by default and algorithm should be allowed to show or hide them. Or perhaps displayed/hidden initially at discretion of the user (e.g., for entering edge weights for MST and SP algorithms).
* *Note: you almost never want both labels and weights of an edge or node in the same algorithm:*
  + *some algorithms want to set labels on both but don’t use weights,*
  + *some want weights on both (usually edge weights from the user, node weights from the program), but no labels,*
  + *some want labels on nodes and weights on edges (or possibly vice versa). Default should be to show neither unless requested by algorithm or by the user (e.g., toggle to show weights only if the user wants to edit them).*
* Current code for drawing labels (GraphPanel) and weights is a rat’s nest with lots of magic numbers. Should have two simple methods: drawNodeLabel and drawEdgeLabel (ditto for weights but there will be a fair amount of code in common). First organize existing code around these. Then use the trick of creating a filled rectangle (oriented “normally”) and drawing text. Size of rectangle should be determined by (i) font characteristics as extracted from Preferences module, (ii) padding – set as a constant somewhere, and (iii) length of string in pixels (should be able to get this directly from the font. Initially can put all labels/weights in the middle of an edge with labels, if present, on top of weights.
* ***Making edges thicker and/or colorable.*** For example: Edge is thicker and some default color when selected; user can change thickness or color under preferences. With nodes, default (when selected) might be shading and a border color. No change if marked (that's just a convenient utility for search algorithms, essentially a built-in data structure).
* Different levels of selection, also specified in defaults – 0 = not selected, 1 = default selection, 2 through *k* as defined by user in preferences. Programmer has API functions, setSelected(0), …, setSelected(*k*), but you can still retain the existing boolean method
* Allow user (or programmer) to choose the size of vertices. This makes sense if displaying really large graphs, e.g., for crossing minimization algorithms.
* Offer choice about size of vertex and whether or not an id will be displayed so that algorithms can be visualized on a large number of nodes (e.g., like DagDisplay but in motion).
* Have a “fit to display window” option that scales the graph so that all the vertices and edges appear inside the display window; this could interact with an absolute scaling mechanism that allows a user to zoom in to an important part of the graph
* Allow user to reposition labels/weights of edges by sliding them along an edge.

### Algorithm

* ***Don’t run algorithm to completion before animation starts.*** Instead, run only until the next step, but still allow backtracking with respect to *display states*; backtracking the complete algorithm state would be impractical; this becomes a little dicey if display and logical attributes are intertwined as in Dijkstra’s algorithm, where the “selected” property (used primarily for display) drives the logic.
* Allow an algorithm to define what’s visible at the outset rather than relying on the user to provide the correct settings (the latter may not even be aware of what they are). For example, there could be methods *setLabelsVisible()* and *setWeightsVisible()*, each with two boolean arguments, one for nodes and one for edges.
* Include a query mechanism that asks the user to enter text and/or select some part of the graph (node/edge).
* A mechanism for changing the order of an adjacency list *while the algorithm is in progress*. This would allow a demo of, e.g., DFS to illustrate the effect that such a reordering has on the classification of edges. The mechanism would allow a backtrack-reorder-forward sequence. The main issue here is user interaction: how does the user specify the reordering and how is a reordering whose dependence is in the past get handled (error message?)
* Current state of an algorithm can be saved, but it appears that the coloring is not preserved, only the labels and weights.
* The ability to declare an arbitrary number of lists of nodes or edges without digging into Java details.
* Ability to skip steps or simply treat the animation as a movie progressing at a given speed. There are multiple options/features for algorithm execution:
  + set breakpoints instead of or in addition to steps
  + control speed by adding sleep between steps
  + allow arbitrary midstream edits

There may be implementation tradeoffs between some of these and the ability to backtrack.