Animation programmers can write algorithms in notation that resembles textbook pseudocode. Java-like syntax is used throughout, e.g., node methods are called using `v.m()`, where `v` is an object of type `Node` and `m` is a method. The `graph` object is implicit – so `graph.m()` is the same as `m()`.

Central to the Galant API is the `graph` object: currently all other parts of the API refer to it. The components of a graph are declared to be of type `Node` or `Edge` and can be accessed/modified via a variety of functions/methods. When an observer or explorer interacts with the animation they move either forward or backward one step at a time. All aspects of the graph API therefore refer to the current *state of the graph*, the set of states behaving as a stack. API calls that change the state of a node or an edge automatically generate a next step, but the programmer can override this using a `beginStep()` and `endStep()` pair. For example, the beginning of our implementation of Dijkstra's algorithm looks like

```
beginStep();
for_nodes(node) {
    node.setWeight( INFINITY );
    nodePQ.add( node);
}
endStep();
```

Without the override this initialization would require the observer to click through multiple steps (one for each node) before getting to the interesting part of the animation.

Functions and macros of the API are listed below.

1 Graph methods

Each node and edge has a unique integer id. The id's are assigned consecutively as nodes/edges are created and may not be altered. The id of a node or edge can be accessed via the `getId()` method.

- `setDirected(boolean b)` causes the graph to be displayed and treated as directed (`b = true`) or undirected (`b=false`); this function can only be applied once at the beginning of the animation. It is useful, e.g., for minimum spanning tree algorithms (`b = false`) or for depth-first search to find strongly connected components (`b = true`).
- `getNodes()` returns a list of nodes of the graph; the return type is the Java templated `List<Node>`.*
- `getEdges()` returns a list of edges of the graph; return type is `List<Edge>`
- `for_nodes( node )` and `for_edges( edge )` circumvent the need to deal with Java templated list objects. Their syntax is:
  `for_nodes( node ) {` *block of code that uses* `node }` (Note: `node` need not be declared.)
  and the corresponding for `for_edges`.
- `getNodeById(id)` returns the node with the given integer id or null if none exists; the only use in current animations is `getNodeById(0)` to specify node 0 (the first node created) as the start node for an algorithm that require one.
- `getEdgeById(id)` returns the edge with the given integer id or null if none exists; not currently used.
- `select(Node n)` makes node `n` the sole highlighted/selected node
- `Node addNode()` returns a new node and adds it to the list of nodes; the id is one greater than the largest id so far; attributes take on default values; weight, label, and position are absent and must be set explicitly by appropriate method calls.
- `addEdge(Node source, Node target)` adds an edge from the source to the target (source and target are immaterial when graph is undirected). There is also a variation with integer arguments that represent the id's of the nodes. As in the case of nodes, the edge is added to the list of

*This will be replaced by a Galant-native equivalent in a future version.

edges and its weight and label are absent.

2 Node and edge methods

Nodes and edges have 'getters' and 'setters' and setters for a variety of attributes, i.e., `set`*a*`([`*a*'*s* `type] x)` and [*a*'*s* type] `get`*a*`)`. In addition there are the usual methods one would expect from a graph API.

Logical attributes: functions and macros

From a node's point of view we would like information about the adjacent nodes and incident edges. The relevant *methods* require the use of Java generics, but macros are provided to simplify graph API access. The macros, which have equivalents in GDR, are:

- `for_adjacent(Node x, Edge e, Node y){` *statements* `}` executes the list of statements for each edge incident on `x`. The statements can refer to `x`, or `e`, the current incident edge, or `y`, the other endpoint of `e`.
- `for_outgoing(Node x, Edge e, Node y){` *statements* `}`
  behaves like `for_adjacent` except that, when the graph is directed, it iterates only over the edges whose source is `x` (it still iterates over all the edges when the graph is undirected).
- `for_incoming(Node x, Edge e, Node y){` *statements* `}`
  behaves like `for_adjacent` except that, when the graph is directed, it iterates only over the edges whose sink is `x` (it still iterates over all the edges when the graph is undirected).

The actual API methods hiding behind these macros are (these are Node methods):

- `List<Edge> getIncidentEdges()` returns a list of all edges incident to this node, both incoming and outgoing.
- `List<Edge> getOutGoingEdges()` returns a list of edges directed away from this node (all incident edges if the graph is undirected).
- `List<Edge> getIncomingEdges()` returns a list of edges directed toward this node (all incident edges if the graph is undirected).
- `Node travel(Edge e)` returns the other endpoint of `e`.

The logical edge methods are:

- `setSourceNode(Node)` and `Node getSourceNode()`
- `setDestination(Node)` and `Node getDestinationNode()`
- `getOtherEndPoint(Node u)` returns `v` where this edge is either `uv` or `vu`.

Nodes and edges also have a mechanism for setting (and getting) arbitrary attributes of type Integer, String, and Double. the relevant methods are `setIntegerAttribute(String key,Integer value)` to associate an integer value with a node and `Integer getIntegerAttribute(String key)` to retrieve it. String and Double attributes work the same way as integer attributes. These are useful when an algorithm requires arbitrary information to be associated with nodes and/or edges. Note that each individual node may have different attributes.

Geometric attributes

Currently, the only geometric attributes are the positions of the nodes. The edges are all straight lines and positions of labels are fixed. The relevant methods for nodes are `int getX()`, `int getY()`, and `Point getPosition()` for the 'getters'. To set a position, one should use either `setPosition(Point)` or `setPosition(int,int)`. However, once a node has an established position, it is possible to change only one coordinate using `setX(int)` or `setY(int)`.

Display attributes

Each node and edge has both a (double) weight and a label. The weight is also a logical attribute in that it is used implicitly as a key for sorting and priority queues. The label is simply text and may be interpreted however the programmer chooses. Both weight and label have values to indicate their absence: `Double.NaN` (not a number) in the case of weights, `null` in the case of labels. In either case the weights/labels are not displayed. Aside from the setters and getters: `setWeight(double)`, `double getWeight()`, `setLabel(String)`, and `String getLabel()`, the programmer can also manipulate and test for the absence of weights/labels using `clearWeight()` and `boolean hasWeight()`, and the corresponding methods for labels.

Nodes can either be plain, highlighted (selected), marked (visited) or both highlighted and marked. Being highlighted alters the the boundary (color and thickness) of a node (as controlled by the implementation), while being marked affects the fill color. Edges can be plain or selected, with thickness and color modified in the latter case.

The relevant methods are:

- `highlight()`, `unHighlight()`, and `boolean isHighlighted()`
- correspondingly, `setSelected(true)`, `setSelected(false)`, and `boolean isSelected()`
- `mark()`, `unMark()`, and `boolean isMarked()`
- correspondingly, `setVisited(true)`, `setVisited(false)`, and `boolean isVisited()`

Although the specific colors for displaying selected nodes or edges are predetermined, the animation implementation can modify the color of a node boundary or an edge, thus allowing for many different kinds of highlighting.[†] The `setColor` and `getColor` methods use String arguments using the RGB format #RRGGBB; for example, the string `#0000ff` is blue.

Of the attributes listed above, weight, label, color and position can be accessed and modified by the user as well as the program. In all cases, modifications by execution of the animation are ephemeral — the graph returns to its original state after execution.

3 Additional programmer information

One awkward feature of Galant's implementation is that global variables must be declared `final`. For arrays and other structures this is not a problem, except for the annoyance of the syntax. So, for example,

```
final int [] theArray = new int[ graph.getNodes.size() ];
```

will work as expected: the entries of `theArray` can be modified as the animation progresses. Not so with scalars. The workaround is along the lines of

```
class GlobalVariables {
    public int myInt;
    public double myDouble;
}
final GlobalVariables globals = new GlobalVariables();
```

and then the globals need to be referred to as `globals.myInt` and `globals.myDouble`, respectively.

[†]It would also be desirable to be able to alter the thickness of edges or node boundaries to make color changes more visible.