1 Galant Programmer Documentation (for version 6.0)

What follows is self-contained documentation for Galant animators (animation programmers). There are many examples included with this distribution, both in the Algorithms directory and in the subdirectories under Research. Some basic examples are in an Appendix in the texhnical report, 2016-Galant-Stallmann.pdf.

Contents

List of Tables

Animation programmers can write algorithms in notation that resembles textbook pseudocode in files that have a .alg extension. The animation examples have used procedural syntax for function calls, as in, for example, setWeight(v,0). Java (object oriented) syntax can also be used: v.setWeight(0). A key advantage of Galant is that a seasoned Java programmer can not only use the Java syntax but can also augment Galant algorithms with arbitrary Java classes defined externally, using import statements. All Galant code is effectively Java, either natively, or via macro preprocessing.

The text panel provides a crude editor for algorithms (as well as GraphML descriptions of graphs); its limited capabilities make it useful primarily for fine tuning and error correction. The animator should use a program editor such as Emacs or Notepad++ (in Java mode) to edit algorithms offline, not a major inconvenience – it is easy to reload algorithms when they are modified without exiting Galant. The Galant editor is, however, useful in that it provides syntax highlighting of Galant functions and macros.

The source code for an algorithm begins with any number (including none) of global variable declarations and function definitions. The animator can import code from other sources using appropriate import statements; these must occur at the very beginning. The code for the algorithm itself follows, starting with the keyword algorithm. A *code block* is a sequence of statements, each terminated by a semicolon, just as in Java. An animation program has the form

> *global variable declarations*
>
> *function definitions*
>
> algorithm {
>   *code block*
> }

Declarations of global variables are also like those of Java:
  type variable_name;
or
  type [] *variable_name*;
to declare an array of the given type. All variables must be initialized either within a function definition or in the algorithm. Unlike Java variables, they cannot be initialized in the statement that declares them.[1] The Java incantation
  type variable_name = new *type*[*size*]
is used to initialize an array with *size* elements initialized to null or 0. Arrays use 0-based indexing: the largest index is $size - 1$. Detailed information about function declarations is in Section 1.2 below.

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

[1] This restriction applies to global variables only. Variables local to function definitions or to the algorithm can be initialized in-line, just as in Java.

Without the beginStep/endStep 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. For convenience the function step() is a synomym for endStep(); beginStep(). If a step takes longer than 5 seconds, the program is terminated under the presumption that there may be an infinite loop.

Functions and macros for the graph as a whole are shown in Table 1, while Table 2 lists some algorithm functions not related to any aspect of a graph.

Note: *The functions/methods provided by Galant may have multiple synonyms for convenience and backward compatibility. A full list of methods and functions is given in Algorithm.java in the subdirectory src/edu/ncsu/csc/Galant/algorithm.*

The nodes and edges, of type Node and Edge, respectively, are subtypes/extensions of GraphElement. Arbitrary attributes can be assigned to each graph element. In the GraphML file these show up as, for example,

<node *attribute_1="value_1"* ... *attribute_k="value_k"* />

Each node and edge has a unique integer id. The id's are assigned consecutively as nodes/edges are created; they may not be altered. The id of a node or edge can be accessed via the id() function. Often, as is the case with the depth-first search algorithm, it makes sense to use arrays indexed by node or edge id's. Since graphs may be generated externally and/or have undergone deletions of nodes or edges, the id's are not always contiguous. The functions nodeIds() and edgeIds() return the size of an array large enough to accommodate the appropriate id's as indexes. So code such as

```
Integer myArray[] = new Integer[nodeIds()];
for_nodes(v) { myArray[id(v)] = 1; }
```

is immune to array out of bounds errors.

1.1 Node and edge methods

Nodes and edges have 'getters' and 'setters' for a variety of attributes, i.e.,
set$a$($\langle a's$ type$\rangle$ x)
and
$\langle a's$ type$\rangle$ get$a$(), where a is the name of an attribute such as Color, Label or Weight. A more convenient way to access these standard attributes omits the prefix get and uses procedural syntax: color(x) is a synonym for x.getColor(), for example. Procedural syntax for the setters is also available: setColor(x,c) is a synonym for x.setColor(c). In the special cases of color and label it is possible to omit the set (since it reads naturally): color(x,c) instead of setColor(x,c); and label(x,c) instead of setLabel(x,c).

1.1.1 Logical attributes: functions and macros

Nodes. 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 are borrowed from their equivalents in GDR) are:

- for_adjacent(x, e, y){ *code block* } executes the statements in the code block 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`. The macro assumes that x has already been declared as Node but e and y are declared automatically.
- for_outgoing(Node x, Edge e, Node y){ *code block* }
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){ *code block* }
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).

List⟨Node⟩ getNodes() NodeSet getNodeSet()	returns a list or set of the nodes of the graph; type NodeSet is built into Galant – see Table 6, but the templated List has not yet been replaced with NodeList
List⟨Edge⟩ getEdges() EdgeSet getEdgeSet()	returns a list of edges of the graph; return types are analogous to those for nodes
for_nodes(v) { *code block* }	equivalent to for (Node v : nodes()) { *code block* }; the statements are executed for each node v
for_edges(e) { *code block* }	analogous to for_nodes
Integer numberOfNodes()	returns the number of nodes
Integer numberOfEdges()	returns the number of edges
int id(Node v), int id(Edge e)	returns the unique identifier of v or e
int nodeIds(), int edgeIds()	returns the largest node/edge identifier plus one; useful when an array is to be indexed using node/edge identifiers, since these are not necessarily contiguous
source(Edge e), target(Edge e)	returns the source/target of edge e, sometimes called the (arrow) tail/head or source/destination
Integer degree(Node v) Integer indegree(Node v) Integer outdegree(Node v)	the number of edges incident on v, total, incoming and outgoing; if the graph is undirected, the outdegree is the same as the degree
EdgeList edges(Node v) EdgeList inEdges(Node v) EdgeList outEdges(Node v)	returns a list of v's incident, incoming or outgoing edges, respectively; outgoing edges are the same as incident edges if the graph is undirected
Node otherEnd(Edge e, Node v) Node otherEnd(Node v, Edge e)	returns the node opposite v on edge e; if v is the source otherEnd returns the target and vice-versa
NodeList neighbors(Node v)	returns a list of nodes adjacent to v
for_adjacent(v, e, w) { *code block* } for_incoming(v, e, w) { *code block* } for_outgoing(v, e, w) { *code block* }	for_adjacent executes the code block for each edge e incident on v, where w is otherEnd(e,v); v must already be declared but e and w are declared by the macro; the other two are analogous for incoming and outgoing edges
getStartNode()	returns the first node in the list of nodes, typically the one with smallest id; used by algorithms that require a start node
isDirected()	returns true if the graph is directed
setDirected(boolean directed)	makes the graph directed or undirected depending on whether `directed` is true or false, respectively
Node addNode() Node addNode(Integer x, Integer y)	returns a new node and adds it to the list of nodes; the id is the smallest integer not currently in use as an id; attributes such as weight, label and position are absent and must be set explicitly by appropriate method calls; the second version puts the node at position (x,y), where x and y are pixel coordinates.
addEdge(Node source, Node target) addEdge(int sourceId, int targetId)	adds an edge from the source to the target (source and target are interchangeable when graph is undirected); the second variation specifies id's of the nodes to be connected; as in the case of adding a node, the edge is added to the list of edges and its weight and label are absent
deleteNode(Node v)	removes node v and its incident edges from the graph
deleteEdge(Edge e)	removes edge e from the graph

Table 1: Functions and macros that apply to the structure of a graph.

print(String s)	prints s on the console; useful for debugging
display(String s)	writes the string s at the top of the window
String getMessage()	returns the message currently displayed on the message banner
error(String s)	prints s on the console with a stack trace; also displays s in popup window with an option to view the stack trace; the algorithm terminates and the user can choose whether to terminate Galant entirely or continue interacting
beginStep(), endStep(), step()	any actions between a beginStep() and an endStep() take place atomically, i.e., all in a single "step forward" action by the user; step() is a synonym for endStep(); beginStep()
Node getNode(String message)	pops up a window with the given message and prompts the user to enter the identifier of a node, which is returned; if no node with that id exists, an error popup is displayed and the user is prompted again
Edge getEdge(String message)	pops up a window with the given message and prompts the user to enter the identifiers of two nodes, the endpoints of an edge, which is returned; if either id has no corresponding node or the the two nodes are not connect by an edge (in the right direction if the graph is directed), an error popup is displayed and the user is prompted again
Node getNode(String p, NodeSet s, String e) Edge getEdge(String p, EdgeSet s, String e)	variations of getNode and getEdge; here p is the prompt, s is the set from which the node or edge must be chosen and e an error message if the node/edge does not belong to the specified set; useful when wanting to specify an adjacent node or an outgoing edge
String getString(String message) Integer getInteger(String message) Double getReal(String message)	analogous to getNode and getEdge; allows algorithm to engage in dialog with the user to obtain values of various types; getDouble is synonymous with getReal
Boolean getBoolean(String message) Boolean getBoolean(String message, String yes, String no)	similar to getString, etc., but differs in that only user input is a mouse click or the Enter key and the algorithm steps forward immediately after the query is answered; the second variation specifies the text for each of the two buttons – default is "yes" and "no"
Integer integer(String s) Double real(String s)	performs conversion from a string to an integer/double; useful when parsing labels that represent numbers
windowWidth(), windowHeight()	current width and height of the window, in case the algorithm wants to rescale the graph

Table 2: Utility functions.

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

- List⟨Edge⟩ edges(v) returns a list of all edges incident to v, both incoming and outgoing.
- List⟨Edge⟩ outgoingEdges(v) returns a list of edges directed away from v (all incident edges if the graph is undirected).
- List⟨Edge⟩ incomingEdges(v) returns a list of edges directed toward v (all incident edges if the graph is undirected).
- Node otherEnd(e, v) returns the endpoint, other than v, of e.

The following are node functions with procedural syntax.

- degree(v), indegree(v) and outdegree(v) return the appropriate integers.
- otherEnd(v, e), where v is a node and e is an edge returns node w such that e connects v and w; the programmer can also say otherEnd(e, v) in case she forgets the order of the arguments.
- neighbors(v) returns a list of the nodes adjacent to node v.

Edges. The logical attributes of an edge e are its source and target (destination) accessed using source(e) and target(e), respectively.

Graph Elements. Nodes and edges both have a mechanism for setting (and getting) arbitrary attributes of type Integer, String, and Double. the relevant methods are listed below. Note that the type can be implicit for the setters – the compiler can figure that out, but needs to be explicit for the getters – in Java, two methods that differ only in their return type are indistinguishable. In each case g stands for a graph element (node or edge).

- set(g, String *attribute*, ⟨*type*⟩ *value*), where *type* can be String, Boolean, Integer, or Double.
- set(g, String *attribute*); the attribute is assumed to be Boolean, the value is set to true.
- String getString(g, String *attribute*)
- Boolean getBoolean(g, String *attribute*)
- Boolean is(String *attribute*), a synonym for getBoolean
- Integer getInteger(g, String *attribute*)
- Double getDouble(g, String *attribute*)

An object oriented syntax can also be used – this is especially natural in case of is, as in v.is("inTree") – see boruvka in the Algorithms directory. These are useful when an algorithm requires arbitrary information to be associated with nodes and/or edges. The user-defined attributes may differ from one node or edge to the next. For example, some nodes may have a depth attribute while others do not.

1.1.2 Geometric attributes

Currently, the only geometric attributes are the positions of the nodes. Unlike GDR, the edges in Galant are all straight lines and the positions of their labels are fixed. The relevant methods for nodes – using procedural syntax – are int getX(Node), int getY(Node) and Point getPosition(Node) for the getters. To set a position, one should use

setPosition(Node,Point)

or

setPosition(Node,int,int).

Once a node has an established position, it is possible to change one coordinate at a time using setX(Node,int) or setY(Node,int). Object-oriented variants of all of these, e.g., v.setX(100), are also available.

The user is allowed to move nodes during algorithm execution and the resulting positions persist after execution terminates. Node position is the only attribute that can be "edited" at runtime. For some animations, however, such as sorting, the animation itself needs to move nodes. To avoid potential conflicts between position changes inflicted by the user and those desired by the animation.

RED	= "#ff0000"
BLUE	= "#00ff00"
GREEN	= "#0000ff"
YELLOW	= "#ffff00"
MAGENTA	= "#ff00ff"
CYAN	= "#00ffff"
TEAL	= "#009999"
VIOLET	= "#9900cc"
ORANGE	= "#ff8000"
GRAY	= "#808080"
BLACK	= "#000000"
WHITE	= "#ffffff"

Table 3: Predefined color constants.

the function movesNodes(), called at the beginning of an algorithm will prevent the user from moving nodes.

1.1.3 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. The conversion functions integer(String) and real(String) – see Table 2 – provide a convenient mechanism for treating labels as objects of class Integer or Double, respectively. The second argument of label (or single argument of the object-oriented setLabel) is not the expected String but Object; any type of object that has a Java toString method will work – numbers have to be of type Integer or Double rather than int or double since the latter are not objects in Java.[2] Thus, conversion between string labels and numbers works both ways.

Aside from the setters and getters: setWeight(double), Double getWeight(), setLabel(Object) 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. The procedural variants in this case are

- setWeight(Node,double),
- Double getWeight(Node) or the more natural Double weight(Node),
- setLabel(Node,Object) or the more natural label(Node,Object),
- getLabel(Node) or the more natural String label(Node)

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 (here Element refers to either a Node or an Edge):

- highlight(Element), unhighlight(Element) and Boolean isHighlighted(Element)
- correspondingly, setSelected(true), setSelected(false), and boolean isSelected()
- mark(Node), unmark(Node) and Boolean isMarked(Node), equivalently Boolean marked(Node).

Although the specific colors for displaying the outlines of nodes or the lines representing edges are predetermined for plain and highlighted nodes/edges, the animation implementation can modify these colors, thus allowing for many different kinds of highlighting. The getColor and setColor methods and their procedural variants have String arguments in the RGB format #RRGGBB; for example,

[2] Galant functions return objects, Integer or Double, when return values are numbers for this reason.

the string `#0000ff` is blue. Here, as in the case of label, color(g) and color(g,c) can be used in place of getColor(g) and setColor(g,c), respectively. Galant defines several color constants for convenience – these are listed in Table 3 – so one can say, e.g., color(g,TEAL) instead of color(g,"#009999").

Note: In the graph display *highlighting takes precedence over color*; if a node is highlighted, its color is ignored and the default highlight color is used.

Special handling is required when any attribute is nonexistent or has a null value – these two are equivalent. When displayed in the graph window, nonexistent labels and weights simply do not show up while nonexistent colors are rendered as thin black lines (thickness determined by user preference). In an animation program, however, nonexistent attributes are handled differently.

- color() returns null as expected
- all functions returning Boolean values, such as highlighted(), marked() and those for attributes defined by the animator, return false
- label() returns an empty string; this ensures that it is always safe to use a label in an expression calling for a string
- weight() throws an exception; there is no obvious default weight; a program can test for the presence/absence of a weight using the hasWeight() or hasNoWeight() methods

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 (of display attributes – recall that node positions are an exception), modifications during runtime are ephemeral – the graph returns to its original, pre-execution, state after running the animation. The user can save the mid-execution state of the graph: select the Export option on the file menu of the *graph* window.

A summary of functions relevant to node and edge attributes (their procedural versions) is given in Table 4.

1.1.4 Global access for individual node/edge attributes and graph attributes

It is sometimes useful to access or manipulate attributes of nodes and edges globally. For example, an algorithm might want to hide node weights entirely because they are not relevant or hide them initially and reveal them for individual nodes as the algorithm progresses. These functionalities can be accomplished by hideNodeWeights or hideAllNodeWeights, respectively. A summary of these capabilities is given in Table 5.

1.2 Definition of Functions/Methods

A programmer can define arbitrary functions (methods) using the construct

function *[return_type] name* (*parameters*) {
    code block
}

The behavior is like that of a Java method. So, for example,

```
function int plus( int a, int b ) {
    return a + b;
}
```

is equivalent to

```
static int plus( int a, int b ) {
    return a + b;
}
```

The *return_type* is optional. If it is missing, the function behaves like a void method in Java. An example is the recursive function visit in depth-first search.

function visit(Node v) { *code* }

id(*element*)	returns the unique identifier of the node or edge
source(Edge e), target(Edge e)	returns the source/target of edge e, sometimes called the (arrow) tail/head or source/destination
mark(Node v), unmark(Node v)	shades the interior of a node or undoes that
Boolean marked(Node v)	returns true if the node is marked
highlight(*element*), unhighlight(*element*)	makes the node or edge highlighted, i.e., thickens the border or line and makes it red / undoes the highlighting
Boolean highlighted(*element*)	returns true if the node or edge is highlighted
select(*element*), deselect(*element*) selected(*element*)	synonyms for highlight, unhighlight and highlighted
Double weight(*element*) setWeight(*element*, double weight)	get/set the weight of the element
showWeight(*element*), hideWeight(*element*) Boolean weightIsVisible(*element*) Boolean weightIsHidden(*element*)	make the weight of the element visible/invisible, query their visibility; weights of the element type have to be globally visible – see Table 5 – for showWeight to have an effect
String label(*element*) label(*element*, Object obj)	get/set the label of the element, the Object argument allows an object of any other type to be converted to a (String) label, as long as there is a toString method, which is true of all major classes (you have to be careful, for example, to use Integer instead of int)
showLabel(*element*), hideLabel(*element*) Boolean labelIsVisible(*element*) Boolean labelIsHidden(*element*)	analogous to the corresponding weight functions
hide(*element*), show(*element*) Boolean hidden(*element*) Boolean visible(*element*)	makes nodes/edges disappear/reappear and tests whether they are visible or hidden; useful when an algorithm (logically) deletes objects, but they need to be revealed again upon completion
String color(*element*) color(*element*, String c) uncolor(*element*)	get/set/remove the color of the border of a node or line representing an edge; colors are encoded as strings of the form "#RRBBGG", the RR, BB and GG being hexadecimal numbers representing the red, blue and green components of the color, respectively; see Table 3 for a list of predefined colors; when an element has no color, the line is thinner and black
boolean set(*element*, String key, ⟨*type*⟩ value)	sets an arbitrary attribute, key, of the element to have a value of a given type, where the type is one of Integer, Double, Boolean or String; in the special case of Boolean the third argument may be omitted and defaults to true; so set(v,"attr") is equivalent to set(v,"attr",true); returns true if the element already has a value for the given attribute, false otherwise
boolean clear(*element*, String key)	removes the attribute key from the element; if the key refers to a Boolean attribute, this is logically equivalent to making it false
⟨*type*⟩ get⟨*type*⟩(*element*, String key) Boolean is(*element*, String key)	returns the value associated with key or null if the graph has no value of the given type for key, i.e., if no set(String key, ⟨*type*⟩ value) has occurred; in the special case of a Boolean attribute, the second formulation may be used; the object-oriented syntax, such as e.is("inTree"), sometimes reads more naturally

Table 4: Functions that query and manipulate attributes of individual nodes and edges. Here, *element* refers to either a Node or an Edge, both the type and the formal parameter.

Boolean nodeLabelsAreVisible() Boolean edgeLabelsAreVisible() Boolean nodeWeightsAreVisible() Boolean edgeWeightsAreVisible()	returns true if node/edge labels/weights are *globally* visible, the default state, which can be altered by hideNodeLabels(), etc., defined below
hideNodeLabels(), hideEdgeLabels() hideNodeWeights(), hideEdgeWeights()	hides all node/edge labels/weights; typically used at the beginning of an algorithm to hide unnecessary information; labels and weights are shown by default
showNodeLabels(), showEdgeLabels() showNodeWeights(), showEdgeWeights()	undoes the hiding of labels/weights
hideAllNodeLabels() hideAllEdgeLabels() hideAllNodeWeights() hideAllEdgeWeights()	hides all node/edge labels/weights even if they are visible globally by default or by showNodeLabels(), etc., or for individual nodes and edges; in order for the label or weight of a node/edge to be displayed, labels/weights must be visible globally and its label/weight must be visible; initially, all labels/weights are visible, both globally and for individual nodes/edges; these functions are used to hide information so that it can be revealed subsequently, one node or edge at a time
showAllNodeLabels() showAllEdgeLabels() showAllNodeWeights() showAllEdgeWeights()	makes all individual node/edge weights/labels visible if they are globally visible by default or via showNodeLabels(), etc.; this undoes the effect of hideAllNodeLabels(), etc., and of any individual hiding of labels/weights
clearNodeLabels(), clearEdgeLabels() clearNodeWeights(), clearEdgeWeights()	gets rid of all node/edge labels/weights; this not only makes them invisible, but also erases whatever values they have
showNodes(), showEdges()	undo any hiding of nodes/edges that has taken place during the algorithm
NodeSet visibleNodes() EdgeSet visibleEdges()	return the set of nodes/edges that are not hidden
clearNodeMarks() clearNodeHighlighting() clearEdgeHighlighting()	unmarks all nodes, unhighlights all nodes/edges, respectively
clearNodeLabels() clearNodeWeights() clearEdgeLabels() clearEdgeWeights()	erases labels/weights of all nodes/edges; useful if an algorithm needs to start with a clean slate with respect to any of these attributes
clearAllNode(String attribute) clearAllEdge(String attribute)	erases values of the given attribute from all nodes/edges, a generalization of clearNodeLabels, etc.
boolean set(String attribute, ⟨*type*⟩ value)	sets an arbitrary attribute of the graph to have a value of a given type, where the type is one of Integer, Double, Boolean or String; in the special case of Boolean the second argument may be omitted and defaults to true; so set("attr") is equivalent to set("attr",true); returns true if the graph already has a value for the given attribute, false otherwise
⟨*type*⟩ get⟨*type*⟩(String attribute) Boolean is(String attribute)	returns the value associated with attribute or null if the graph has no value of the given type for attribute, i.e., if no set(String attribute, ⟨*type*⟩ value) has occurred; in the special case of a Boolean attribute, the second formulation may be used
clearAllNode(String attribute) clearAllEdge(String attribute)	erases the value of the given attribute for all nodes/edges

Table 5: Functions that query and manipulate graph node and edge attributes globally, i.e., for all nodes or edges at once. Also included are functions that deal with graph attributes.

NodeList and EdgeList: lists of nodes or edges, respectively. We use *list* as shorthand for either NodeList or EdgeList, *type* for either Node or Edge and *element* for Node v or Edge e.

Initialization

list L = new *type*List();	creates an empty list L

Other methods

type first(*list* L)	returns the first element of L
type get(int i, *list* L)	returns the i-th element of L; indexing is 0-based, so get(0,L) is the same as first(L)
add(*element* x, *list* L)	adds the element x to the end of list L; along with first we get the effect of a queue
remove(*element* x, *list* L)	removes the first occurrence of x from L

NodeQueue and EdgeQueue: queues of nodes or edges, respectively. Same conventions as for lists.

void enqueue(*element*)	adds the element to the rear of the queue
type dequeue()	returns and removes the element at the front of the queue; returns null if the queue is empty
type remove()	returns and removes the element at the front of the queue; throws an exception if the queue is empty
type element()	returns the element at the front of the queue without removing it; throws an exception if the queue is empty
type peek()	returns the element at the front of the queue without removing it; returns null if the queue is empty
size()	returns the number of elements in the queue
isEmpty()	returns true if the queue is empty

NodeStack and EdgeStack: stacks of nodes or edges, respectively.

void push(*element*)	adds the element to the top of the stack
type pop()	returns and removes the element at the top of the stack; throws an exception if the stack is empty
type peek()	returns the element at the top of the stack without removing it; returns null if the stack is empty
size(), isEmpty()	analogous to the corresponding queue methods

NodeSet and EdgeSet: sets of nodes or edges, respectively.

boolean add(*element*)	adds the element to the set; returns true if the element was a new addition, false otherwise
boolean remove(*element*)	removes the element returns true if the element was present, false otherwise
boolean contains(*element*)	returns true if the element is in the set
size(), isEmpty()	analogous to the corresponding methods for other data structures

Table 6: Built-in data structures and their methods. These methods use object-oriented syntax: $\langle structure\rangle.\langle method\rangle(\langle arguments\rangle)$ and are created using, e.g., NodeQueue Q = new NodeQueue(); the new operator in Java.

1.3 Data Structures

Galant provides some standard data structures for nodes and edges. These are described in detail in Tables 6. Data structures use object-oriented syntax. For example, to add a node v to a NodeList L, the appropriate syntax is L.add(v).

1.4 Sorting, Priority Queues, and Comparators

Priority queues of nodes and edges are also provided by Galant – see Table **??**. Closely related is the sort macro, which translates to the Java Collections.sort method. By default, sorting and priority queues use the weight of a node or edge as its key and sorting is by increasing weight, while priority queues are min-heaps – lowest weight item is extracted using the removeMin method. However, this can be overriden if a comparator for a different attribute is provided when sort is called or when the priority queue is created. The other attributes can be of type String, Integer or Double – see earlier discussion in Section 1.1 and the appropriate functions in Table 4. Comparators for them are generated with calls to functions of the form get*type*Comparator, where *type* is String, Integer or Double. There is an optional second argument, which, if true causes the comparisons to be the reverse of the usual order, i.e., decreasing for numbers and a max-heap instead of a min-heap.

In case of a priority queue the type of heap and the attribute used for comparison

A notable exception is decreaseKey for priority queues. The operation pq.decreaseKey(v,k) does pq.remove(v) followed by pq.add(v). The new key is implicit – it is the weight of the element. In practice, assuming pq is a NodePriorityQueue, v is a Node and k is a Double, the sequence would be

```
setWeight(v, k);
pq.decreaseKey(v,k);
```

which would translate to

```
setWeight(v, k);
pq.remove(v);
pq.add(v)
```

As pointed out earlier, the weight of a node or edge is used for priority in a priority queue or for sorting. The programmer can change this default as well as the fact that the nodes/edges prioritized by increasing weight (the priority queue is a min-heap).

The functionality of a priority queue depends on how it is initialized via a constructor. Taking EdgePriorityQueue as an example (NodePriorityQueue is analogous), the standard initialization, one that defines a min-heap that uses weights as key is
EdgePriorityQueue q = new EdgePriorityQueue()
or the declaration
EdgePriorityQueue q
could be separate from
q = new EdgePriorityQueue()

Variations on the constructor call (part following new) are

- EdgePriorityQueue(true) to create a max-heap based on weights (if the argument is false the result is a min-heap, the default)
- EdgePriorityQueue("attribute"), where "attribute" is a numerical attribute of an edge
- EdgePriorityQueue("attribute", true) to create a max-heap with the given attribute.

[A workaround to the fact that the "attribute" constructors don't work is a setAttribute() method]

1.5 Queries

An animation program can query the user for various kinds of input. For example, the interactive_dfs algorithm asks the user to give a starting node for a (directed) depth-first search and to give another start node if the search terminates before all nodes are visited. The different query options are listed in Table 2.

A query statement in an animation program initiates an algorithm step unconditionally, i.e., even if it occurs within a beginStep-endStep pair. After the user responds to the query she has to do another step forward before the animation proceeds (except in case of a Boolean query). If the user steps backward after responding to a query, the query is not invoked again. Subsequent forward steps use the same answer. Thus it is not possible to allow a user to explore multiple alternative executions in the same run (such a feature would require major enhancements to the existing implementation).

Queries for nodes and edges ask for node id's (two of them in case of an edge). Galant checks whether an id is that of a valid node and, in case of an edge, whether an edge between the two nodes exists. If the graph is currently directed, the direction of the edge also has to correspond. Any violation causes an exception to be thrown – a popup window reports the nature of the error and allows the user to choose (a) different id(s). The animator can impose additional restrictions by specifying a set of permissible nodes/edges (unvisited nodes in the case of interactive_dfs). If so, the animator also specifies an error message in case the additional restriction is violated.

Other queries allow an animation to get strings, Boolean values, or numbers from the user. Examples are in the binary_tree and grid algorithms which create complete binary trees or grid graphs based on tree height or grid dimensions, respectively, specified as integers by the user. These queries work the same way as those for vertices and edges. In case of numbers Galant checks whether the input string is a valid integer or floating point number and reports an error otherwise.

Boolean queries are a special case. The user does not type a response. The only options are to press one of two buttons with the mouse or to press the Enter key to specify the default answer (true). The animator can specify the text displayed on the buttons; defaults are "yes" and "no". Another difference with Boolean queries is that the algorithm steps forward immediately when the user responds to the query.[3]

1.6 Exceptions: Compile and Runtime Errors

Errors can occur at compile time, either because a macro is malformed or because the Java code, after macro translation, has errors. The reporting of Java compiler errors is straightforward. They are reported, with line numbers, on the console. The line numbers correspond to those in the Java code listing that also appears on the console (even if there are no errors). In almost all cases the line numbers also correspond to those of the original algorithm (before macro translation) in the text window.[4]

Errors due to malformed macros are not, unfortunately, reported with line numbers. To make matters worse, unbalanced parentheses or braces inside a function definition or one of the for_... macros result in a malformed macro exception. The best strategy is to use a program editor that does automatic indentation.

Runtime errors are also reported with (almost always correct) line numbers. Galant makes every effort to catch errors before they result in, for example, null pointer exceptions in the Galant implementation. Every function with a graph element argument checks that the argument is not null and reports a GalantException if it is. The second or third line in the stack trace refers to the point in the algorithm where the exception occurred. All exceptions, whether those caught as GalantException's with meaningful messages or those caught in the Galant impemmentation code, result in

[3]The reason this is not the case with other queries is that errors may need to be handled before the algorithm can proceed. Synchronization between the query and the algorithm is not straightforward.

[4]The only known exception is a function definition where the parameters are placed on multiple lines.

message	type/source	explanation
programmer message m	runtime	animation program has encountered an error(m) call
programmer message m	runtime	user selected a node or edge not in the set speficied by a query of the form getNode/getEdge(p, S, m), where p is the prompt, S the set, and m the error message
Nonexistent node or edge	runtime	a node/edge is null or does not exist in current state
Graph element has no weight	runtime	no weight was given, neither during editing nor earlier during execution
No edge with source v and target w exists	runtime	can occur when user responds to a query for an edge or the algorithm asks for a specific edge using getEdge(v, w)
Empty graph	runtime	animation attempts to get a node when there is none
No node with id i exists	runtime	called getNodeById(i) when no node with id i exists
Node has been deleted	runtime	called getNodeById(i) when node has been deleted
Attempt to removeMin from max heap (or vice versa)	runtime	priority queues can be initialized as either min heaps or max heaps; Galant checks to make sure the correct remove method is used
Attempt to add null node/edge to (priority) queue	runtime	what it says, prevents later problem with null element
Node/edge has no attribute a when attempting to add to priority queue	runtime	Attribute a, which was specified as the one to use for comparisons when the priority queue was initialized, is not present for the element; the default attribute is weight
Unable to compute center for node	any time	something got messed up with the x and y coordinates (or the layer information in case of a layered graph), e.g., with a setPosition call
Something went wrong when processing algorithm block	macro processing	probably some unbalanced parens, braces or brackets in the algorithm
Missing *right paren, bracket, brace* in *code* ...	macro processing	what it says; the *code* shows only the beginning of the code block in which the error occurred
macro name: Curly braces required	macro processing	missing curly braces after a function header
No compiler found, need a JDK	compiler	either no JDK is installed or JAVA_HOME is not set up correctly
Invalid tab - use untitled graph or untitled algorithm	text editor	attempt to save a file when there's something wrong with the current tab/panel (should not happen)
No text when invoking GraphMLParser	GraphML parser	attempting to parse empty file
Missing id for node	GraphML parser	no id specified for the node; nodes are required to have id's; they are optional for edges
Bad id	GraphML parser	the id of a node or edge is not an integer
Duplicate id i	GraphML parser	there is more than node/edge with id i
Missing source/target	GraphML parser	an edge has no source/target in its GraphML representation
Bad source/target id	GraphML parser	the source/target specified in the GraphML file is not a legal integer
Source/target node missing	GraphML parser	the integer id of the source/target does not correspond with any node
Bad weight	GraphML parser	the weight of a node/edge is not a valid floating point number
Bad x/y-coordinate	GraphML parser	the x or y coordinate of a node is not a legal integer
Missing/bad layer/positionInLayer	GraphML parser	something is wrong with layer or positionInLayer of a node in a layered graph

Table 7: Galant exceptions.

a stack trace on the console and a popup window. The latter allows to user to choose whether to continue, meaning that the algorithm is terminated and Galant returns to edit mode, or exit from Galant completely. Null pointer exceptions can almost always be tracked down by looking for the first reference to the algorithm in the stack trace, an item of the form

Galant.algorithm.code.compiled.*algorithm_name*.run(*algorithm_name*:*line_number*)

Exceptions can also occur in edit mode: when reading a graph from a file, when specifying a node or edge after a keyboard shortcut, or when giving the weight of a node or edge. A complete list of Galant exceptions is in Table 7.