## 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 technical report, 2016-Galant-Stallmann.pdf.

### Contents


**1 Galant Programmer Documentation (for version 6.0) . . . 1**
- 1.1 Node and edge attributes and methods . . . 6
  - 1.1.1 Logical attributes: functions and macros . . . 6
  - 1.1.2 Geometric attributes . . . 7
  - 1.1.3 Display attributes . . . 8
  - 1.1.4 Global access for individual node/edge attributes and graph attributes . . . 9
  - 1.1.5 Functions that behave as declarations . . . 12
- 1.2 Definition of Functions/Methods . . . 12
- 1.3 Data Structures . . . 13
- 1.4 Sorting, Priority Queues, and Comparators . . . 15
- 1.5 Queries . . . 17
- 1.6 Exceptions: Compile and Runtime Errors . . . 18


### List of Tables


- 1 Functions and macros that apply to the structure of a graph. . . . 4
- 2 Utility functions. . . . 5
- 3 Predefined color constants. . . . 8
- 4 Functions that query and manipulate attributes of individual nodes and edges. . . . 10
- 5 Functions that query and manipulate graph node and edge attributes globally. . . . 11
- 6 Functions that act as declarations. . . . 12
- 7 Basic Galant data structures and initialization. . . . 13
- 8 Operations on Galant data structures (procedural versions). . . . 14
- 9 Priority queue initialization and functions in Galant. . . . 16
- 10 Galant exceptions. . . . 19

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.

Some Java constructs have no equivalent procedural syntax. Most notably *sOne*.equals(*sTwo*), where *sOne* and *sTwo* are strings (of type String) is the only means of equality comparison for strings. Equality of numbers is tested using the == operator. To compare strings lexicographically you have to use *sOne*.compareTo(*sTwo*), which returns a positive number, negative number, or 0 depending on whether *sOne* lexicographically follows, precedes or is equal to *sTwo*, respectively .

Descriptions in this document give the most natural and most common constructs for procedures and functions, usually procedural rather than object-oriented. A perusal of the file Algorithm.java in directory
src/edu/ncsu/csc/Galant/algorithm
shows the animator the variety of synonyms for these constructs as well as functions not described here. For the interested Java programmer, object-oriented syntax appears as a call in the body of the corresponding function/procedure. For example,

```
public void mark(Node n) throws Terminate, GalantException {
  checkGraphElement(n);
  n.mark();
}
```

shows how the procedural version of mark translates to the corresponding Node method. Here, checkGraphElement(n) throws an exception with a meaningful error message if n is null. The Terminate exception results when the user chooses to exit the animation (not really an exception so much as an alert to the animation program).

We encourage an animator to browse the example animations in subdirectories Algorithm, Test and in the subdirectories of Research.

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*;
to declare a variable 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 $size - 1$. The *type* prefix is omitted if the array has been declared globally. For example, you could have a global declaration

String [] alpha;
and then, within a function or the algorithm body,

alpha = new String[10]. The array alpha would then contain 10 null strings (not to be confused with "")[2] indexed from 0 through 9.

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);
    insert(node, pq);
}
endStep();
```

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 synonym 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 $x$ can be accessed via the function call id($x$). 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.[3] The functions nodeIds() and edgeIds() return

[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.

[2] For example, alpha[2].equals("") would cause a null pointer exception but not if you did alpha[2] = "abc" first – then it would simply return false.

[3] The edges in GraphML files are not required to have id's. If they have none, id's are assigned as the file is parsed.

| | |
|---|---|
| NodeList getNodes()<br>NodeSet getNodeSet() | returns a list or set of the nodes of the graph; see Section 1.3 for more information about the return types |
| EdgeList getEdges()<br>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 : getNodes() ) { *code block* }; the statements in *code block* are executed for each node v |
| for_edges(e) { *code block* } | analogous to for_nodes with getEdges() in place of getNodes() |
| 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)<br>Integer indegree(Node v)<br>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)<br>EdgeList inEdges(Node v)<br>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 |
| NodeList neighbors(Node v)[a] | returns a list of nodes adjacent to v |
| Node otherEnd(Edge e, Node v)<br>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 |
| for_adjacent(v, e, w) { *code block* }<br>for_incoming(v, e, w) { *code block* }<br>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 |
| NodeList visibleNeighbors(Node v)<br>EdgeList visbleEdges(Node v)<br>EdgeList visbleInEdges(Node v)<br>EdgeList visbleOutEdges(Node v) | return lists of neighbors and edges like the corresponding functions (without the visible modifier) defined above; here only the visible nodes/edges are returned – see Table 4 |
| 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()<br>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)<br>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 |

[a]The functions edges, inEdges, outEdges, and neighbors have equivalents that return sets: edgeSet, incomingSet, outgoingSet and neighborSet, respectively.

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 message s as a banner 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,<br>NodeSet s,<br>String e)<br>Edge getEdge(String p,<br>EdgeSet s,<br>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 s; useful when wanting user to specify an adjacent node or an outgoing edge |
| String getString(String message)<br>Integer getInteger(String message)<br>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)<br>Boolean getBoolean(String message,<br>String yes, String no) | similar to getString, etc., but differs in that the 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)<br>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 re-scale the graph |

Table 2: Utility functions.

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 attributes and 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($x$,$c$).

In the subsections below we discuss attributes in more detail, using the categories articulated in the GDR paper – (earlier) citation in the technical report.

#### 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. Macros that hide some Java syntax and extra function calls are provided for that purpose. The macros (which are borrowed from their equivalents in GDR) are:

- for_adjacent($v, e, w$) { *code block* }
  executes the statements in the code block for each edge incident on node $v$. The statements can refer to $v$, or $e$, the current incident edge, or $w$, the other endpoint of $e$. The macro assumes that $v$ has already been declared as Node but it declares $e$ as Edge and $w$ as Node automatically.
- for_outgoing($v, e, w$) { *code block* }
  behaves like for_adjacent except that, when the graph is directed, it iterates only over the edges whose source is $v$ (it still iterates over all the edges when the graph is undirected).
- for_incoming($v, e, w$) { *code block* }
  behaves like for_adjacent except that, when the graph is directed, it iterates only over the edges whose target (destination) is $v$ (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):

- EdgeList edges($v$) returns a list of all edges incident to $v$, both incoming and outgoing.
- EdgeList outgoingEdges($v$) returns a list of edges directed away from $v$ (all incident edges if the graph is undirected).
- EdgeList 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.

---

These assigned id's are not preserved when the graph (GraphML file) is saved.

**Graph Elements.** Nodes and edges both have a mechanism for setting (and getting) arbitrary attributes of type Integer, String, and Double. the relevant functions 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*, $\langle type \rangle$ *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($g$, 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.alg in the Algorithms directory.

Arbitrary attributes are useful when an algorithm requires additional 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.

**Note:** *Attributes retain their types throughout algorithm execution but Galant does not attempt to guess the type of an attribute when reading a graph from a file.* For example, suppose an algorithm does set(v, "pre", 5). To read the value of attribute pre for node v the algorithm will have to use getInteger(v, "pre") (or one of its synonyms) – getDouble(v, "pre") or getString(v, "pre") will return null. However, suppose the user exports the graph to a file mine.graphml after setting the attribute. The GraphML representation for node v will have pre="5", but when Galant reads mine.graphml, the attribute pre, and any other user-defined attribute, will be treated as a string attribute. So getInteger(v, "pre") will return null whereas getString(v, "pre") will return the string "5". This is not a problem unless the user wants to save the state of an algorithm execution midstream *and then use it as a starting point.* The only workaround is for an animation program to do its own parsing, using the utility functions integer and real – see Table 2.

### 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 functions are int getX(Node $v$), int getY(Node $v$) and Point getPosition(Node $v$) for the getters. The Java type/class Point has fields x and y that can be retrieved using $p$.x and $p$.y, where $p$ is a Point. To set a position, use

    setPosition(Node $v$, Point $p$)

or

    setPosition(Node $v$, int $x$, int $y$).

Once a node has an established position, it is possible to change one coordinate at a time using setX(Node $v$, int $x$) or setY(Node $v$, int $y$).

The user is allowed to move nodes during algorithm execution and the resulting positions persist after execution terminates (other attributes do not). Node position is the only attribute that can be "edited" by the user 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, the function movesNodes(), called at the beginning of an algorithm, will keep the user from moving nodes (mouse actions on the graph panel have no effect).

| | |
|---|---|
| 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.

### 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 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.[4] Thus, conversion between string labels and numbers works both ways.

The display and print functions also take arbitrary objects as arguments. So display($g$) and print($g$), where $g$ is a node or edge will print information about the attributes of $g$ (as a list in square brackets). Often, you only want to display/print the id of a node or the source and target of an edge. The appropriate incantations for display are display(id($g$)) and display(string($g$)), respectively. The string function is designed specifically for this purpose and applies only to edges.

In order to make the display of weights more attractive, weights that happen to be integers are shown without the decimal point and at most two positions to the right of the decimal point are shown.

Aside from the setters and getters: setWeight(GraphElement $g$, double $wt$), Double getWeight($g$), label($g$, Object $o$) and String label($g$), the programmer can also manipulate and test for the absence of weights/labels using clearWeight($g$) and Boolean hasWeight($g$), and the corresponding methods for labels. It is also possible to remove an arbitrary attribute $a$ using clear($g$, String $a$), but there is not yet a has function for arbitrary attributes.[5] You have to test whether the getter returns null.

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 (thickness is controlled by user preference), 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 $g$ refers to an object of type GraphElement, a Node or an Edge):

- highlight($g$), unhighlight($g$) and Boolean highlighted($g$)
- correspondingly, select($g$), deselect($g$), and Boolean selected($g$)
- mark(Node $v$), unmark(Node $v$) and Boolean marked(Node $v$)

Although the specific colors for displaying the outlines of nodes or the lines representing edges are

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

[5] Semantic details, such as whether to implement hasString($g$, $a$), hasInteger($g$, $a$), etc., and/or to use has($g$, $a$) to check whether the attribute exists in any form need to be worked out.

predetermined for plain and highlighted nodes/edges, the animation implementation can modify colors explicitly, thus allowing for many different kinds of highlighting. Use the color($g$) (getter), color($g$, $c$) (setter) and uncolor($g$) functions, where $g$ is a node/edge and $c$ a color string in the RGB format `#RRGGBB`; for example, the string `#0000ff` is blue. 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 one of the native Galant attributes 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($g$) returns null as expected
- all functions returning Boolean values, such as highlighted($g$), marked(Node $v$) and those for attributes defined by the animator, return false
- label($g$) returns an empty string; this ensures that it is always safe to use a label in an expression calling for a string
- weight($g$) throws an exception; there is no obvious default weight; a program can test for the presence/absence of a weight using the hasWeight($g$) or hasNoWeight($g$) methods

Of the attributes listed above, weight, label, color and position can be accessed and modified by the user as well as the program. Except in case of node positions, as noted above, the user can do this only in edit mode. In all cases of display attributes 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 final display attribute is visibility. An algorithm sometimes involves deletion of nodes or edges, or, as is the case with our implementation of Boruvka's algorithm, some edges are no longer important. Graph elements can be hidden and made to reappear using the hide and show functions.

To understand how showing and hiding of nodes and edges works, we need to distinguish between the logical state of a node/edge and what is displayed. Logically, for any graph element $g$, show($g$) makes $g$ visible and hide($g$) makes it invisible. For nodes, the logical and display state are the same – a logically visible node will be drawn on the graph window. But an edge will be drawn only if it is logically visible *and both of its endpoints are also logically visible.*

A summary of functions relevant to node and edge attributes (their procedural versions) is given in Table 4. Some of the most important functions, those relevant to the structure of the graph, are listed in Table 1.

### 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 animation might want to hide node weights initially and reveal them for individual nodes as they become relevant. Or it may want to hide edges individually as it progresses to get the effect of deleting them and then reveal them again at the end. These functionalities can be accomplished by hideAllNodeWeights[6] or showEdges, respectively. A summary of these capabilities is given in Table 5.

- An algorithm hides some edges individually and uses showEdges to make all of them visible in one step later.
- An algorithm hides some nodes individually (along with their incident edges); it later makes

[6] The modifier All distinguishes show/hideAllNode/EdgeLabels/Weights from show/hideNode/EdgeLabels/Weights. The latter are designed to mimic user toggle buttons – see Section 1.1.5.

Here, *element* refers to either a Node or an Edge, both as a type and as a formal parameter.

| | |
|---|---|
| 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 |
| string(Edge e) | returns a string of the form "($s$,$t$)", where $s$ = source(e) and $t$ = target(e) |
| 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 |
| NodeList unmarkedNeighbors(Node v) | returns a list of the adjacent nodes that are not 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*)<br>selected(*element*) | synonyms for highlight, unhighlight and highlighted |
| Double weight(*element*)<br>setWeight(*element*, double weight) | get/set the weight of the element |
| showWeight(*element*), hideWeight(*element*)<br>Boolean weightIsVisible(*element*)<br>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*)<br>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*)<br>Boolean labelIsVisible(*element*)<br>Boolean labelIsHidden(*element*) | analogous to the corresponding weight functions |
| hide(*element*), show(*element*)<br>Boolean hidden(*element*)<br>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; if node is hidden, all its incident edges are hidden as well; show($v$), where $v$ is a node, will show only the incident edges that are not hidden |
| String color(*element*)<br>color(*element*, String c)<br>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)<br>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.

These functions are designed to access or manipulate attributes for all nodes or edges at once instead of individually. Also included are functions that deal with graph attributes.

| | |
|---|---|
| hideAllNodeLabels()<br>hideAllEdgeLabels()<br>hideAllNodeWeights()<br>hideAllEdgeWeights() | hide all node/edge labels/weights these functions are typically used to hide information so that it can be revealed subsequently, one node or edge at a time |
| showAllNodeLabels()<br>showAllEdgeLabels()<br>showAllNodeWeights()<br>showAllEdgeWeights() | make all individual node/edge weights/labels visible this undoes the effect of hideAllNodeLabels(), etc., and of any individual hiding of labels/weights |
| clearNodeLabels(), clearEdgeLabels()<br>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(), showGraph() | undo any hiding of nodes/edges that has taken place during the algorithm; showNodes() only shows edges currently (logically) visible; showGraph() restores visibility of both nodes and edges |
| NodeSet visibleNodes()<br>EdgeSet visibleEdges() | return the set of nodes/edges that are not hidden |
| clearNodeMarks()<br>clearNodeHighlighting()<br>clearEdgeHighlighting() | unmarks all nodes, unhighlights all nodes/edges, respectively |
| clearNodeLabels()<br>clearNodeWeights()<br>clearEdgeLabels()<br>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)<br>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)<br>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)<br>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.

| | |
|---|---|
| movesNodes() | keeps the *user* from moving nodes during animation execution |
| setDirected(boolean $d$) | determine whether the animation will treat the graph as directed ($d =$ true) or undirected ($d =$ false) |
| showNodeLabels(), showEdgeLabels()<br>showNodeWeights(), showEdgeWeights() | ensure that node/edge weights/labels will be visible throughout animation execution, unless explicitly hidden by the animation, or visibility is changed by the user using the toggle buttons |
| hideNodeLabels(), hideEdgeLabels()<br>hideNodeWeights(), hideEdgeWeights() | ensure that the specified attributes remain hidden throughout the animation unless visibility is changed using toggle buttons; the attributes are hidden *regardless of whether the animation shows or hides them* |

Table 6: Functions that act as declarations.

all nodes visible and selectively makes some of their incident edges visible; nodes that were not hidden are unaffected; the visibility of any edges is not affected either.

- An algorithm hides individual nodes and edges and later restores visibility to the whole graph; the function to use in this case is showGraph.

#### 1.1.5 Functions that behave as declarations

Some Galant functions are independent of animation steps and are intended as declarations at the beginning of the algorithm code. These are listed in Table 6. We already mentioned movesNodes() – it signals that the animation will control node movement and prevents the user from moving nodes.

A second example is setDirected. It has the same function as the toggle buttons at the top of the graph window. In some cases, such as shortest paths algorithms and breadth-first search, it makes sense for the user to decide whether or not the animation should treat the graph as a directed graph. Other animations, such as dfs_d, are specifically designed for directed graphs. In those cases it makes sense for the animation to make the graph directed. When setDirected is called, the toggle buttons (not visible during execution) are set as well, so that the state of the buttons is consistent with that of the graph.

The remaining declarations involve visibility of weights and labels. As with directedness, these functions mimic the actions of the appropriate toggle buttons. The user may also change visibility of labels and weights during execution. The declarations allow the animation to make visible exactly those attributes that are relevant during its execution. For example, the weighted graph algorithms (shortest paths and minimum spanning trees) need the edge weights to be visible; a showEdgeWeights() call at the beginning ensures that this happens regardless of whether the user made edge weights visible during editing (or execution of a previous algorithm).

### 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;
}
```

In the table below *EdgeCollection* refers to any preexisting data structure whose elements are edges while *NodeCollection* refers to one whose elements are nodes. Each type of structure can be initialized as empty or as containing elements of a given collection. As with arrays – see page 3 – the new operator creates a new instance of a structure.

| data structure | initializers |
|---|---|
| EdgeList | new EdgeList()<br>new EdgeList(*EdgeCollection*) |
| NodeList | new NodeList()<br>new NodeList(*NodeCollection*) |
| EdgeSet | new EdgeSet()<br>new EdgeSet(*EdgeCollection*) |
| NodeSet | new NodeSet()<br>new NodeSet(*NodeCollection*) |
| EdgeQueue | new EdgeQueue()<br>new EdgeQueue(*EdgeCollection*) |
| NodeQueue | new NodeQueue()<br>new NodeQueue(*NodeCollection*) |
| EdgePriorityQueue | new EdgePriorityQueue()<br>new EdgePriorityQueue(*EdgeCollection*) |
| NodePriorityQueue | new NodePriorityQueue()<br>new NodePriorityQueue(*NodeCollection*) |

Table 7: Basic Galant data structures and initialization.

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* }

### 1.3 Data Structures

Galant provides some standard data structures for nodes and edges. These are described in detail in Tables 7 and 8. All Galant data structures are instances of the Java Collection interface and therefore automatically have the methods required of a collection. For the interested Java programmer, they are also extensions of relevant concrete Java classes and inherit those methods as well. For details, see the Java source code in directory
src/edu/ncsu/csc/Galant/graph/datastructure.

Galant provides procedural versions of the required methods size, isEmpty, add and remove for all structures, i.e., if $C$ is a collection (data structure) and $g$ is a graph element of the appropriate type, the functions are size($C$), empty($C$), add($e$, $C$) and remove($e$, $C$), respectively.

Table 7 shows how each data structure can be initialized in two different ways. One is to create a collection with no elements, the other creates one from another collection that has the same element type. For example, if S is a previously declared and initialized EdgeSet (with elements added initially or later) you can say

EdgeList L = new EdgeList(S)

This becomes especially useful for priority queues – see Section 1.4 and Table 9 below for more details on initializing and using these. To create a priority queue containing all edges you would

**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.

| | |
|---|---|
| *type* first(*list* $L$) | returns the first element of $L$ |
| *type* top(*list* $L$) | returns the first element of $L$ (top of stack) |
| add(*element* $g$, *list* $L$) | adds the element $g$ to the end of list $L$ |
| push(*element*, *list* $L$) | adds the element $g$ to the front of list $L$ (pushes on stack) |
| *type* pop(*list* $L$) | removes and returns the first element of $L$ |
| remove(*element*, *list* $L$) | removes the first occurrence of $g$ from $L$ |
| size(*list* $L$) | returns the number of elements in $L$ |
| empty(*list* $L$) | returns true if $L$ is empty |

**NodeQueue and EdgeQueue:** queues of nodes or edges, respectively. Same conventions as for lists, except we use *queue* in place of *list*.

| | |
|---|---|
| void put(*element*, *queue*) | adds the element to the rear of the queue; throws an exception if the element is null |
| *type* get(*queue*) | returns and removes the element at the front of the queue; throws an exception if the queue is empty |
| *type* front(*queue*) | returns the element at the front of the queue without removing it; throws an exception if the queue is empty |

**NodeSet and EdgeSet:** sets of nodes or edges, respectively; same conventions as for lists, except for the use of *set*. The first two table entries give natural syntax for set membership.

| | |
|---|---|
| boolean *set*.contains($g$) | returns true if $g$ is an element of the set, where $g$ is a node or edge, as appropriate; if $g$ has the wrong type, this method will simply return false |
| boolean $g$.in(*set*) | returns true if $g$ is an element of the set, where $g$ is a node or edge, as appropriate; here, if $g$ has the wrong type, the error will be caught by the compiler |
| union($s_1$, $s_2$) | returns the union of $s_1$ and $s_2$; here $s_1$, $s_2$ and the return value are all of type NodeSet or EdgeSet |
| intersection($s_1$, $s_2$) | returns the intersection of $s_1$ and $s_2$ |
| difference($s_1$, $s_2$) | returns the set difference $s_1 - s_2$, elements that are in $s_1$ but not in $s_2$ |
| symmetricDifference($s_1$, $s_2$) | returns the set symmetric difference of $s_1$ and $s_2$, i.e., the union of $s_1 - s_2$ and $s_2 - s_1$ |
| subset($s_1$, $s_2$) or $s_1$.subset($s_2$) | returns true if $s_1$ is a subset of $s_2$, false otherwise |

Table 8: Operations on Galant data structures (procedural versions).

simply say

EdgePriorityQueue PQ = new EdgePriorityQueue(getEdges())

Java collections also have iterators. The syntax for accessing all elements of a data structure is

for ( *type* $g$ : *collection* ) { *code body* }

where *type* is either Node or Edge, $g$ is a variable, and *collection* is a data structure containing elements of the given type. The code body is executed for each element in the collection, referenced as $g$. For example, if S is an EdgeSet you could do

```
for ( Edge e : S ) {
   highlight(e);
}
```

to highlight all the edge in S. The order of appearance of the elements depends on the structure. For lists and queues it is the order in which they were added, for sets it is random (the elements are hashed), and for priority queues it is based on the values of the keys.

It is possible to create structures of type EdgeStack and NodeStack but if you want simple procedural syntax and friendlier error reporting we recommend using lists instead.

Table 8 summarizes the operations available on each of the data structures. The semantics of functions add, remove, size, and empty are shown in the table only for lists, but apply to the others as well.

### 1.4 Sorting, Priority Queues, and Comparators

Priority queues are a special case in that the order of appearance, which affects the semantics of the key functions, is dependent on the *comparator* that is used to determine their total order. Sorting also depends on a comparator, so we discuss sorting and priority queues in the same section. The default comparator for graph elements, nodes and edges, sorts them by increasing weight. If $L$ is a list of nodes or edges

sort($L$)

rearranges the elements of $L$ so that they are in order of increasing weight (if any element has no weight there is a null pointer exception). Galant sort is actually a Galant macro that expands to the Java Collections.sort. A comparator as second argument to sort can alter the default behavior. Galant makes available comparators that specify the attribute (instead of weight) and the ordering (increasing or decreasing) as follows.

- getDoubleComparator(String attribute) returns a comparator based on the values of the given attribute, in increasing order;
  the default comparator is equivalent to getDoubleComparator("weight")
- getDoubleComparator(String attribute, boolean reverse) returns a comparator based on the values of the given attribute, in decreasing order if reverse is true, increasing if it is false;
  the default comparator is equivalent to getDoubleComparator("weight", false).
- getIntegerComparator(String attribute) and getIntegerComparator(String attribute, boolean reverse) behave like getDoubleComparator except that the attribute has integer values;
  for example, getIntegerComparator("id", true) sorts nodes (or edges) by decreasing id.
- getStringComparator(String attribute) and getStringComparator(String attribute, boolean reverse) behave like their numerical counterparts; here the ordering is lexicographic;
  for example, getStringComparator("label", true) sorts nodes or edges based on their labels, in decreasing lexicographic order ("zebra" comes before "antelope").

To do the sorting simply call sort with the appropriate comparator as second argument. For example, after

```
EdgeList edges = getEdges();
sort(edges, getStringComparator("label", true));
```

**Initialization**

| | |
|---|---|
| new EdgePriorityQueue()<br>new NodePriorityQueue() | creates an empty min-heap that uses weight as key |
| new EdgePriorityQueue($C$)<br>new NodePriorityQueue($C$) | creates a min-heap that uses weight as key and contains all the elements of $C$, which can be a list, set, or queue |
| new EdgePriorityQueue(boolean isMax)<br>new NodePriorityQueue(boolean isMax) | creates an empty priority queue that uses weight as key, if isMax is true, the queue is a max-heap, otherwise it's a min-heap |
| new EdgePriorityQueue(Comparator C)<br>new NodePriorityQueue(Comparator C) | creates an empty priority queue that uses C to compare elements; if C is a GraphElementComparator, the attribute and whether it's a min or max heap are extracted from the comparator |

**Functions.** Here $g$ is a GraphElement, i.e., an Edge or Node and $Q$ is a priority queue of the appropriate type. The return type *element* is either Edge or Node.

| | |
|---|---|
| void add($g$, $Q$) | adds $g$ to the priority queue; the priority is defined by the initialization method |
| void insert($g$, $Q$) | same as add but does error checking |
| void remove($g$, $Q$) | removes $g$ from $Q$; used for any element; not efficient because it searches the whole queue |
| *element* best($Q$) | returns the best element, min or max value as determined by initialization, in $Q$; runtime is constant |
| *element* removeBest($Q$) | returns and removes the best element, min or max value as determined by initialization, in $Q$; runtime is $O(\lg \lvert Q \rvert)$ |
| void changeKey($g$, $Q$ | reorganizes $Q$ taking into account a change in value for $g$ of the attribute specified at initialization; *the animation program must execute the actual change beforehand*; runtime is $O(\lvert Q \rvert)$ because this translates to a remove followed by an insert |

**Methods.** These have not (yet) been made available in procedural syntax. Conventions same as for functions

| | |
|---|---|
| *element* $Q$.min()<br>*element* $Q$.max() | equivalent to *element* best($Q$), specific to min/max heap; an exception if heap is the wrong kind, based on initialization |
| *element* $Q$.removeMin()<br>*element* $Q$.removeMax() | equivalent to *element* removeBest($Q$), specific to min/max heap; an exception if heap is the wrong kind, based on initialization |
| $Q$.changeKey($g$, *val*) | equivalent to set($g$, "*a*", *val*), where $a$ is an attribute, followed by changeKey($g$, $Q$); here $a$ must be an attribute with a Double value – otherwise the method has no effect |
| $Q$.changeIntegerKey($g$, *val*) | same as $Q$.changeKey($g$, *val*) except that $a$ must be an attribute with an Integer value |
| $Q$.changeStringKey($g$, *val*) | same as $Q$.changeKey($g$, *val*) except that $a$ must be an attribute with a String value |
| $Q$.decreaseKey($g$, *val*) | same as $Q$.changeKey($g$, *val*); provided because it's standard nomenclature in many algorithms |

Table 9: Priority queue initialization and functions in Galant.

the list edges consists of all edges of the graph, sorted in decreasing lexicographic order by their labels.

**Caution:** *The type of the attribute must match the type of the comparator.* The incantation getIntegerComparator("label") does not work as expected. Because label is not an integer attribute, the values being compared will all be null and any sort or priority queue operation based on the comparator will result in a null pointer exception.

Table 9 shows the procedural versions of priority queue methods made available by Galant, along with the four ways to initialize priority queues. Also listed are object-oriented methods that yield additional functionality not yet implemented in procedural form.

If the priority queue is initialized from an existing data structure $C$ the only option is for it to be a min-heap (the element that can be accessed and removed efficiently is the one with smallest value) based on weight. An empty queue can be initialized as a max-heap (using weight) or as a heap that uses any valid comparator. A GraphElementComparator is a special case that stores both the attribute used for comparison and whether the heap is to be min or max, obtained using the get*type*Comparator functions described earlier. The additional information allows Galant to report errors if an element to be added has a null value for the given attribute or if the animation program attempts to removeMin from a max-heap or vice-versa.

Priority queue functions are usually as efficient as expected. A notable exception is changeKey. The operation changeKey($g$, $Q$) does remove($g$, $Q$) followed by insert($g$, $Q$) and takes O($|Q|$) because of the remove, which needs to search the whole queue.

Unless you use object-oriented syntax you have to change the value of an attribute before calling changeKey. For example, you can do

```
NodePriorityQueue Q = new NodePriorityQueue();
...
Node v = ...;
```

and later

```
setWeight(v, 25);
changeKey(v, Q);
```

or

```
Q.changeKey(v, 25);
```

### 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.[7]

### 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.[8]

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 implementation code, result in 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 list of the most common Galant exceptions is in Table 10.

[7] 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.

[8] 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 specified 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 |
| 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 parentheses, braces or brackets in the algorithm |
| Missing *right parenthesis, 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 10: Galant exceptions.