1 Galant Programmer Documentation

Animation programmers can write algorithms in notation that resembles textbook pseudocode. 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)`.

The source code for an algorithm begins with any number (including none) of global variable declarations and function definitions. This is followed by the code for the algorithm itself, starting with the keyword algorithm. A *code block* is a sequence of statements, each terminated by a semicolon, just as in Java. The main algorithm has the form

```
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. The Java incantation
type variable_name = new *type*[*size*]
is used to initialize an array with *size* elements intialized to null or 0. Arrays use 0-based indexing: the largest index is $size - 1$. Function declarations are described in more detail in Section 1.2.1 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();
```

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.

Functions and macros for the graph as a whole are shown in Table 1. Also included are a few functions that are global to an algorithm without any direct connection to a graph.

The nodes and edges, of type Node and Edge, respectively, are subtypes 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 and 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 accomodate the appropriate id's as indexes. So code such as

```
Integer myArray[] = new Integer[nodeIds()];
```

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
List⟨Node⟩ getNodes() NodeList nodes()	returns a list of the nodes of the graph; type NodeList is built into Galant and equivalent to the Java List⟨Node⟩
List⟨Edge⟩ getEdges() EdgeList edges()	returns a list of edges of the graph; return type EdgeList is analogous to NodeList
for_nodes(v) { *statement; . . .* }	equivalent to for (Node v : getNodes()) { *statement; . . .* }; the statements are executed for each node v
for_edges(e) { *statement; . . .* }	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
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 sets the position of the node at (x,y)

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 or to an algorithm more generally.

```
for_nodes(v) {
    myArray[id(v)] = ...
}
```

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 synomym 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`)`.

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 have equivalents in GDR, are:

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

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

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

The above all use Java syntax, as in `v.travel(e)`. The following are node-related 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 are its source and target (destination).

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

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
`setIntegerAttribute(String key,Integer value)`
to associate an integer value with a node and
`Integer getIntegerAttribute(String key)`
to retrieve it. String and Double attributes work the same way as integer attributes. These are useful when an algorithm requires arbitrary information to be associated with nodes and/or edges. 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 either `setPosition(Node,Point)` or `setPosition(Node,int,int)`. Once a node has an established position, it is possible to change only one coordinate using `setX(Node,int)` or `setY(Node,int)`. Object-oriented variants of all of these, e.g., `v.setX(100)`, are also available.

Ordinarily nodes can be moved by the user during algorithm execution and the resulting positions persist after execution terminates. For some algorithms, such as sorting, the algorithm itself needs to move nodes. It is desirable then to keep the user from moving nodes. The declaration `movesNodes()` at the beginning of an algorithm accomplishes this.

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. Aside from the setters and getters: `setWeight(double)`, `Double getWeight()`, `setLabel(String)` and `String getLabel()`, the programmer can also manipulate and test for the absence of weights/labels using `clearWeight()` and `boolean hasWeight()`, and the corresponding methods for labels. The procedural variants in this case are `setWeight(Node,double)`, `Double weight(Node)`,[1] `label(Node,String)`,[2] and `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 selected nodes or edges are predetermined, the animation implementation can modify the color of a node boundary or an edge, thus allowing for many different kinds of highlighting. The `setColor` and `getColor` methods use String arguments using the RGB format #RRGGBB; for example, the string `#0000ff` is blue. There are several predefined color constants:

[1] The *get* is omitted here for more natural syntax.
[2] A natural syntax that resembles English. However, `setLabel(Node,String)` is also allowed.

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
⟨*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

Table 2: 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.

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

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

A summary of functions relevant to node and edge attributes (their procedural versions) is given in Table 2.

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 3.

1.2 Additional programmer information

A Galant algorithm/program is executed as a method within a Java class. In order to shield the Galant programmer from Java ideosyncrasies, some features have been added.

1.2.1 Definition of Functions/Methods

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

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

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

clearNodeMarks()
clearNodeHighlighting()
clearEdgeHighlighting()
	unmarks all nodes, unhighlights all nodes/edges, respectively

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 3: 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.

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( Node v )` { *code* }
The conversion of functions into Java methods when Galant code is compiled is complex and may result in indecipherable error messages.

1.2.2 Data Structures

Galant provides some standard data strutures for nodes and edges automatically. These are described in detail in Table 4.

NodeQueue and EdgeQueue: queues of nodes or edges, respectively.

void enqueue(Node v), void enqueue(Edge e)	adds v or e to the rear of the queue
Node dequeue(), Edge dequeue()	returns and removes the Node or Edge at the front of the queue; returns null if the queue is empty
Node remove(), Edge remove()	returns and removes the Node or Edge at the front of the queue; throws an exception if the queue is empty
Node element(), Edge element()	returns the Node or Edge at the front of the queue without removing it; throws an exception if the queue is empty
Node peek(), Edge peek()	returns the Node or Edge 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(Node v), void push(Edge e)	adds v or e to the top of the stack
Node pop(), Edge pop()	returns and removes the Node or Edge at the top of the stack; throws an exception if the stack is empty
Node peek(), Edge peek()	returns the Node or Edge at the top of the stack without removing it; returns null if the stack is empty
size(), isEmpty()	analogous to the corresponding queue methods

NodePriorityQueue and EdgePriorityQueue: priority queues of nodes or edges, respectively.

void add(Node v), void add(Edge e)	adds v or e to the priority queue; the priority is defined to be its weight – see Section 1.1.3
Node removeMin(), Edge removeMin()	returns and removes the Node or Edge with minimum weight; returns null if the queue is empty
boolean remove(Node v), boolean remove(Edge e)	removes v or e; returns true if the v or e is present, false otherwise
void decreaseKey(Node v, double key), void decreaseKay(Edge e, double key)	changes the weight of v or e to the new key and reorganizes the priority queue appropriately; since this is accomplished by removing and reinserting the object, i.e., inefficiently, this method can also be used to increase the key
size(), isEmpty()	analogous to the corresponding queue methods

Table 4: Built-in data structures and their methods. These methods use object-oriented syntax: $\langle \mathit{structure} \rangle . \langle \mathit{method} \rangle (\langle \mathit{arguments} \rangle)$ and are created using, e.g., NodeQueue Q = new NodeQueue(); the new operator in Java.